Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Improving Synchronization and Data Access in Parallel Programming Models

Doctoral Dissertation of:
Ettore Speziale

Advisor:
Prof. Stefano Crespi Reghizzi

Tutor:
Prof.ssa Donatella Sciuto

Supervisor of the Doctoral Program:
Prof.ssa Barbara Pernici

2012 - XXV edition

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci 32, I-20133 — Milano

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Improving Synchronization and Data Access in Parallel Programming Models

Doctoral Dissertation of:
Ettore Speziale

Advisor:
Prof. Stefano Crespi Reghizzi

Tutor:
Prof.ssa Donatella Sciuto

Supervisor of the Doctoral Program:
Prof.ssa Barbara Pernici

2012 - XXV edition

Acknowledgements

During the last three years I had the opportunity to study and work in the amazing field of parallel programming. What makes this field unique is the mixture of techniques you have to know: you have to know how hardware works, how to compile/analyze/optimize parallel programs, and how to support their execution at run-time. But what makes this three-years long journey incredible was the companions I found along the way.

First of all, I would really like to thank my advisor, Professor Stefano Crespi Reghizzi. Beside the technical stuffs, I really like to thank him for choosing me for some teaching duties, a task that helped improving my poor communication skills. Second, I would like to thank STMicroelectronics, and, in particular, Diego Melpignano for supporting my Ph.D. studies through a scholarship.

I would also like to thank Professor Eduard Ayguadé and Doctor Vicenç Beltran, who hosted me for a long time at Barcelona Supercomputing Center, and all people I meet there. From the same institution, I also thank Professor Rosa Badia, who served as this dissertation reviewer.

Next, let me thank all people I found in the 127 office ad Dipartimento di Elettronica e Informazione.

From the Formal Languages and Compilers Group, I thank Andrea Di Biagio for teaching me a lot of stuffs about coding and compilers. Moreover, I would like to thank him for the work we did together. I thank Michele Tartara for always supporting me with administrative duties and for many brainstorms we did together. I would like to thank Alberto Magni and Michele Scandale, two exceptional coders that share with me some LLVM-related development. Last, I would like to thank Professor Giampaolo Agosta, the old man of the office, for bearing me for such a long time.

I thank members of the Cryptology group, whit whom I shared the office. In particular I would like to to thank Professor Gerardo Pelosi for its advices full of wisdom, and Doctor Alessandro Barenghi for its fools, but extremely funny, advices.

Finally, I would like to thank my parents and all friends who tolerate my delays and misses to some social activities due to last-time duties, often encountered during Ph.D. studies. Among the other, I would like to

thank Doctor Martina Maggio and Doctor Guido Salvaneschi for helping me with many communication and administrative duties, and for bearing with me for more than just three years.

Sommario

Oggigiorno le architetture parallele sono il principale mezzo utilizzato per sfruttare il crescente numero di transitori disponibili all'interno dei circuiti integrati. Il cambiamento epocale da architetture ottimizzate per l'esecuzione di programmi sequenziali ad architetture pensate per l'esecuzione di codice parallelo è da imputare alla non sostenibilità delle crescenti richieste di potenza elettrica e dall'inabilità del sottosistema di accesso alla memoria di fornire dati al ritmo richiesto dall'unità di esecuzione centrale. D'altronde, l'utilizzo efficiente di multiple unità di esecuzione parallele non è un compito banale. Infatti, per ottenere dei guadagni prestazionali è spesso necessario ottimizzare attentamente le applicazioni, come dimostrato da anni nel campo del calcolo ad alte prestazioni.

Per mascherare tutta questa complessità, i modelli di programmazione parallela espongono una visione semplificata dell'architettura. Invece di mostrare tutte le singole unità parallele, essi si avvalgono di costrutti di alto livello, come ad esempio i cicli paralleli. La traduzione di questi concetti sulle unità di esecuzione parallele è attuato tramite una combinazione di compilatori ottimizzanti e librerie di supporto progettate per guidare l'esecuzione del programma. Tuttavia, data la disponibilità di un grande e variegato numero di architetture parallele, mascherare i dettagli di basso livello spesso limita le prestazioni, che quindi non sono comparabili con quelle ottenibili attraverso una ottimizzazione manuale.

Lo scopo ti questa tesi è analizzare le inefficienze legate all'utilizzo di unità di esecuzione parallele, ed ottimizzarle tramite tecniche da applicare durante l'esecuzione del programma. In particolare si analizza ed ottimizza, l'esecuzione di riduzioni contestualmente ad una operazione di sincronizzazione tramite barriere. Dopodiché, si mostra come l'utilizzo di tecniche dinamiche permette di sfruttare l'affinità tra dati e computazioni per limitare il più possibile la penalità di accesso alla memoria nel contesto di architetture NUMA, dal punto di vista di due modelli di programmazione parallela: OpenMP e MapReduce. Segue quindi una proposta di utilizzo di tecniche leggere di compilazione dinamica per massimizzare l'utilizzo delle architetture parallele, ed infine si analizza la robustezza ai guasti delle primitive di sincronizzazione primitivi, un meccanismo fondamentale utilizzato da ogni programma parallelo.

Abstract

Today, parallel architectures are the main vector for exploiting available die area. The shift from architectures tuned for sequential programming models to ones optimized for parallel processing follows from the inability of further enhance sequential performance due to power and memory walls. On the other hand, efficient exploitation of parallel computing units looks a hard task. Indeed, to get performance improvements it is necessary to carefully tune applications, as proven by years of High Performance Computing using MPI.

To lower the burden of parallel programming, parallel programming models expose a simplified view of the hardware, by relying on abstract parallel constructs, such as parallel loops or tasks. Mapping of those constructs on parallel processing units is achieved by a mix of optimizing compilers and run-time techniques. However, due to the availability of an huge number of very different parallel architectures, hiding low-level details often prevents performance to be comparable with the one of hand-tuned code.

This dissertation aims at analyzing inefficiencies related to the usage of parallel computing units, and to optimize them from the runtime perspective. In particular, we analyze the optimization of reduction computations when performed together with barrier synchronizations. Moreover, we show how runtime techniques can exploit affinity between data and computations to limit as much as possible the performance penalty hidden in NUMA architectures, both in the OpenMP and MapReduce settings. We then observe how a lightweight JIT compilation approach could enable better exploitation of parallel architectures, and lastly we analyze the resilience to faults induction of synchronization primitives, a basic building block of all parallel programs.

Contents

List of Figures

List of Tables

1 Introduction

The switch to parallel general purpose computing is one of the most challenging steps in the history of computer science. Indeed, before 2005, parallelism was mainly exploited at the hardware level, while most of the programming models were based on the abstraction of a single execution flow. Parallel programming was relegated highly specialized application fields, such as High Performance Computing or Internet daemons.

Nowadays, the parallel nature of the hardware is fully exposed to programmers, hence improving performance of applications cannot be resolved by simply upgrading to a newest processor: the major change between two successive generation of processors is not the performance of the single execution unit, but the number of execution units. Thus, a greater number of resources is available, and programmers must correctly exploit them to increase application performance.

Different factors have led to this change in the architecture evolution trend. First of all, exposing a sequential programming model while guaranteeing excellent performance requires to automatically identify independent instructions in the input programs and to map them onto parallel hardware resources. Super-scalar processors perform this operation at run-time, exploiting aggressive *Instruction Level Parallelism* techniques in order to keep the pipeline as full as possible. Combining this with very short pipeline stages allows to boost the core operating frequency, and thus the number of completed instructions per clock cycle. However, these techniques rely on the ability to *detect* parallelism at run-time, which is definitely not an easy task [147]. Moreover, relevant power budgets are necessary [30], and the memory subsystem technologies must be in par with processing units technologies in order to fulfill data requests, which is not the case [150].

From the compiler perspective, exploitation of parallel hardware can be seen as optimization of sequential applications: the compiler is in charge of detecting parallel components in the application, and mapping them onto different processing elements. Due to the limited knowledge of source code achievable by automatic compiler-based analyses, these techniques – e.g. *Decoupled SoftWare Pipelining* [114] – cannot fully exploit all available parallel processing units, thus programming models exposing the true parallel nature of the hardware look better for achiev-

ing maximum performance, as proved by MPI [103] in the domain of large scale HPC applications.

On the other hand, exposing parallel resources to the programmer implies charging him with more responsibilities. For instance, when using the MPI library a programmer must both organize communication between the different processing units, and distribute data across them. To reduce the burden of parallel programming, different programming models focus on hiding these aspects behind high-level constructs, such as *parallel loops*. The compiler, eventually assisted by a runtime, is then in charge of mapping parallel constructs over the available processing units.

A parallel programming model is often based on one of the following parallel computing idioms: i) data-parallelism; ii) task-parallelism; iii) data-flow parallelism. Each of them is specialized in handling a different kind of parallelism.

Data-parallel programming models support regular applications, such as HPC codes. That code is characterized by computing a value out of a set of input matrices. Usually, inputs are analyzed by a means of a *loop* which iterations are almost *independent*. Programmers identify such loops, and tell the compiler that their iterations can be scheduled in parallel. Synchronization inside loop iterations is up to programmers. OpenMP [16] mainly supports parallel loops in the context of shared memory machines. However, since in modern machines the memory access cost is not uniform across different processing units, *Partitioned Global Address Space* languages, like Unified Parallel C [143] and Chapel [44] have been introduced. Together with the identification of parallel loops, they allow programmers specifying how matrices should be distributed across available processing units. This extra information allows the compiler to schedule iterations in order to minimize the cost of accessing the memory. A special case of data-parallelism is represented by languages targeting GPGPUs, like CUDA [6] and OpenCL [87], that allow fine tuning of parallel loop iterations in order to maximize performance. On the other hand, portability is achieved only at the language level, and little changes in the target architecture can exhibit very different performance. Moreover, full exploitation of GPGPUs resources requires non trivial programming efforts.

Task-parallel programming models focus on extracting as many as possible parallelism from applications that do not exhibits a regular behaviour. Indeed, there are no parallel loops to identify. Rather than, at some point, the application spawns independent computations. They are identified by programmers, and called *tasks*. Since at compile-time parallel computations cannot be identified, this approach completely

relies on efficient run-time techniques to manage task lifetimes. Cilk [56] is the best known task-parallel language.

Finally, by specializing the language for a specific application domain, it is possible to completely hide parallelism to programmers. For instance, in the context of signal processing, data-flow programming models allow expressing applications by a means of graphs. Each node represents a small computation, often a filter application. Each arc is labelled with two labels: the label at the source node represents the number of resources produced by running the computation related to the source node, while the label at the target node represents the number of resources needed to trigger the execution of the target node computation. The compiler analyzes the *data-flow graph* and builds a schedule, which steady state can be run in a lock-step fashion by means of multiple processing units. StreamIt [62] is the best known data-flow language.

Thesis Contribution

The main goal of this dissertation is to analyze performance problems hidden in explicit parallel programming models, and optimize them to enable full exploitation of parallel processing units.

Dealing with Synchronization Primitives Embarrassing parallel applications allow efficient parallelization because the work can be partitioned into independent sections, that can be executed without any synchronization by available parallel processing units. However, this kind of application is not very common. Indeed, if multiple processing units contribute in computing something, they must coordinate through synchronization primitives.

In the HPC setting, synchronization is mainly achieved through the *barrier* primitive. Due to its large usage, efficient algorithms have been developed [72] to limit as much as possible the bottleneck it introduces in parallel computations. A barrier synchronization is often associated with the computation of an aggregate value – *reduction*. This results in introducing two bottlenecks in the parallel computation, one due to the barrier synchronization, and the other due to reduction computation.

The barrier execution can be seen as an opportunity to optimize reduction computation. Indeed, Chapter 3 shows how reduction computation can be performed together with the execution of the barrier algorithm, hence paying synchronization cost once. Results of this work have been published in [134].

As a side note, incorrect usage of synchronization primitives is known to be one of the most common error in concurrent, and thus parallel,

programming. Moreover, silicon technologies have reached the point where *soft faults* can happens also inside the integrated circuit rather than only at its boundaries [43], so a key question is how faults insisting on hardware related to the execution of synchronization primitives influence parallel computations. Chapter 7 investigates the problem, by analyzing the robustness to faults of critical sections, either protected by means of mutexes of by exploiting hardware transactional memory. Results of this work have been published in [64].

Efficient Data Access Many parallel architectures include memory subsystems that exhibit different performance depending on which processing unit is requesting the data. Thus, loads and stores must be carefully organized to avoid incurring in expensive accesses. Traditional approaches – e.g. PGAS languages [14, 44, 128, 143] – distribute data structures across the memory subsystems, and then schedule computations to maximize data locality according with the selected distribution.

However, this implies programmers must take into account the distribution of data across available processing units. Chapter 4 faces the problem from a different perspective. Instead of specifying how data structures must be distributed, programmers specify the expect access pattern to the data. This information is exploited by a customized OpenMP runtime to schedule parallel loop iterations on processing units where the expected memory access penalty is minimized. Results of this work have been published in [25].

Efficient data access is also a relevant feature when considering the cost, in term of needed resources, to perform a memory access. In the setting of *Data Intensive Scalable Computing*, tasks access data which is potentially spread across an entire cluster of commodity machines. Executing tasks on machine with no local copy of input data results on accessing disks on a remote machine, thus the resource usage is increased, both in terms of energy requirements, and of network bandwidth.

Chapter 5 introduces a task assignment algorithm that takes into consideration resource usage as well. Given a DISC job, modelled as a set of independent tasks, it keeps under control consumed resources by means of a latency threshold. Indeed, assignments involving expensive data accesses are considered only up to the latency threshold. If the latency goal is not satisfied – e.g. no assignment can be found – consuming more resources does not allow to achieve good latency performance, hence the algorithm assigns left tasks by containing resource consumption.

Runtime Optimization A key feature of parallel computing is the lack of a reference architecture. Indeed, with respect to architectures designed to support sequential programming models, explicitly parallel hardware is shipped with a higher number of configurations. This prevent the construcytion of an exact model of the target architecture, which in turns limits aggressive compiler optimizations.

To offset such problems, some programming models exploit a *Just-In-Time* compiler to delay final code emission just before the application is run. For instance, in CUDA [6] code intended to be run on GPGPU is distributed in form of a byte-code language, while in OpenCL [87] kernel source codes is distributed together with the application.

However, using a JIT compiler incurs in relevant performance penalties, both in terms of raw performance and from the resource usage perspective. Chapter 6 proposes the design of a compilation pipeline that splits the optimization duty between compile-time and run-time. At compile-time, code should be analyzed and code sections suitable for run-time optimization must be detected. Then, instead of employing code specialization, the compiler emits code to *apply* the optimization at run-time. This should avoid using a full-fledged JIT compiler, while keeping some of its features. This work was initially introduced in [52].

Thesis Structure

The rest of this dissertation is organized as follow. Chapter 2 introduces the world of parallel computing, from both the hardware, software, and workload perspectives. Chapter 3 describes the optimization of reductions by means of a special barrier synchronization primitive. Chapter 4 deals with the problem of locality-aware scheduling of OpenMP parallel loop iterations, while Chapter 5 focuses on efficient task assignment in the DISC setting. Chapter 6 introduces a proposal for a lightweight JIT optimization pipeline, and Chapter 7 analyzes the robustness of synchronization primitives to faults. Finally, Chapter 8 concludes.

2 An Overview of Parallel Computing

Today, multi-core technologies are the main vector to increase performance of computer architectures. The shift from sequential to parallel processing has been enforced by physical limits, such as high power budgets and the inability of the memory subsystem to feed high-speed CPUs, which prevent efficient improvements of sequential performance. However, dealing with multi-core architectures is harder than programming single-core architectures, due to the exposure of the parallel hardware to programmers. In this chapter we propose a brief survey about parallel architectures, parallel programming models, and workloads expected to be run on multi-core machines.

2.1 Introduction

After decades of performance improvement guaranteed by boosting single-core capabilities, the computer science discipline has to find alternative ways to continue increasing performance. Indeed, semiconductors physical limits, disparity between improvements in core and memory subsystems, and increased power requests have been identified as the main factors preventing performance improvement of single-core architectures.

An alternative way was found in the context of *multi-core* architectures. Instead of providing architectures built around the concept of a single, powerful core, starting around 2005 architecture designers started providing processors composed by multiple simpler cores. With respect to previous solutions, the performance of the single core is lower, but due to Moore's law effects, the available transistors allow to pack more than one core per die, thus the overall performance of the processor – as a collection of cores – is increased.

However, with respect to single-core technologies, programming multi-core architectures has to face with a wider range of problems. For instance, there is no an unique reference model, hence for each kind of architecture, different programming models and best practices must be employed. Moreover, it is not guaranteed that every algorithm can be

efficiently parallelized – the parallel versions cannot scale, or even worst, the sequential version is more efficient.

Nowadays, parallel processing is a very diversified realm. To correctly exploit the available resources, programmers must be aware of the available parallel architectures. They also need to know the different parallel programming models, and how each application can benefit from parallel hardware. This allow to pursuit the most suitable parallelization strategy, to select the most appropriate programming model, and to target the most suitable architecture.

The rest of this chapter is organized as follow. Section 2.2 deals with parallel computing from the hardware perspective, while Section 2.3 introduces the best known programming models and languages for parallel computing. Section 2.4 briefly depicts the workload expected to be run by parallel hardware. Finally, Section 2.5 concludes.

2.2 The Hardware Perspective

During the entire life of Computer Science as a discipline, hardware has driven the evolution of languages, programming models and software architectures. Hardware has been continuously improved in order to allow programs to run faster.

Language and programming models are built on top of hardware, hence in order to achieve good performance it is needed to select the right language and programming model for each hardware device. The choice must be taken considering hardware features, so, due to the huge amount of available devices, a taxonomy is required to model relevant hardware features.

2.2.1 Flynn's Taxonomy

Hardware can be classified considering its elementary building blocks. In the Flynn's taxonomy [54] each hardware device is modelled as a set of *processing units*, in charge of executing program instructions fetched from an *instruction pool* by means of *instruction streams*. Data is stored in a *data pool*, accessed through *data streams*.

Classification is performed considering the number of processing units, and how they are connected to the instruction pool and to the data pool.

Single Instruction Single Data The architecture is composed by a single processing unit. There is a single data stream feeding the processing unit. This kind of architecture dominates the general purpose market

up to year 2005. Today, it is used as a base building block for designing large multi-core machines, and it is still used in applications were raw computational capabilities is not a strict requisite, such as embedded micro-controllers. A single core desktop processor, like the Intel Pentium, is a representative of this class.

Single Instruction Multiple Data The architecture is composed by multiple processing units. Every processing unit executes the same instruction, but data is fetched from different streams. Vector processors, equipped with vector functional units able to apply the same operation over large arrays, was among the firsts members of this class. Today, GPGPUs multi-processors are the last instances of this class. Indeed, each multi-processor is actually composed by multiple highly-coupled processing units. Each of them executes the same instruction, while referencing a different set of memory locations.

Multiple Instruction Multiple Data The architecture is composed by multiple independent processing units. Each of them executes different instructions, coming from different streams. Data needed by each instruction is fetched from different streams. Modern multi-core processors fall into this category. Indeed, each core is completely independent from the other, and thus it can execute whichever instruction it wants, working on a private data stream.

Multiple Instruction Single Data The architecture is composed by multiple processing units, executing different instructions working on the same data. This class has been defined for symmetry purposes, so finding a representative is not trivial. However, systolic arrays can be interpreted as MISD architectures.

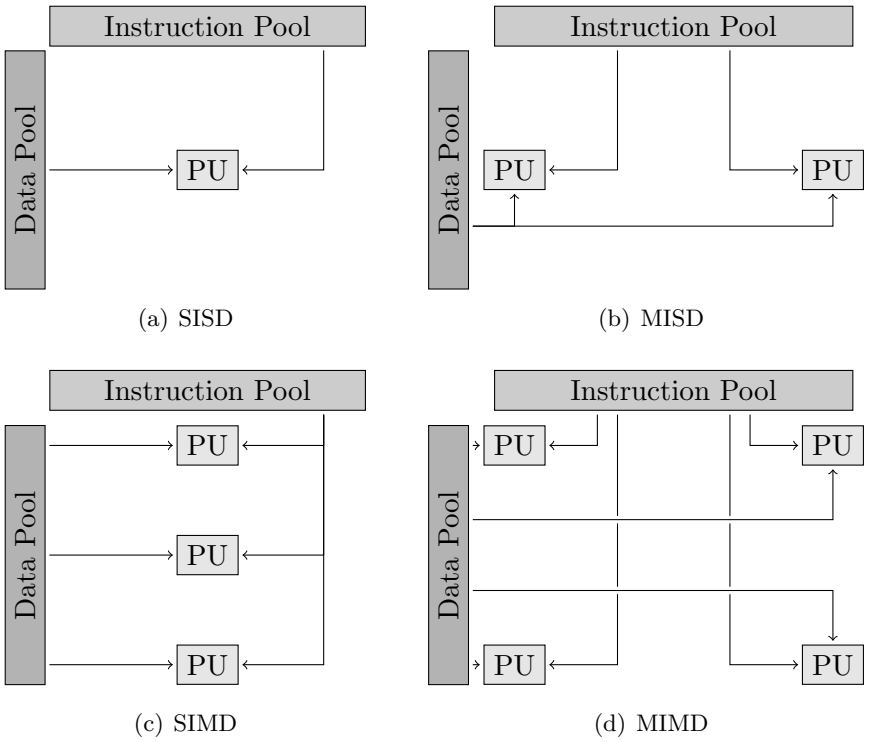
Table 2.1 summarize Flynn's taxonomy, while Figure 2.1 gives a graphical representation of how different Flynn's classes access to instruction and data. All architecture classes must access to both instructions and data, respectively represented by *Instruction Pool* and *Data Pool*. Access is performed by means of streams, represented by arrows connecting processing units to pools.

2.2.2 Traditional Hardware Performance Improvements

Up to 2005, the majority of general-purpose architectures were SISD, thus computer architects focused on efficiently supporting sequential applications.

Table 2.1: Flynn's taxonomy of computer architectures

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD



(a) SISD (b) MISD

(c) SIMD (d) MIMD

Figure 2.1: Interactions of Flynn's taxonomy classes with instruction and data. The fundamental building-block is the processing unit. Classification is based on how processing units access data and instructions through streams

Improvement of sequential performance was initially achieved through extending the architecture in order to implement in hardware some heavily-used functionalities. This in turn fostered the diffusion of *Complex Instruction Set Computing* architectures, which became the dominant ones in the '60 and '70. Typical characteristic of CISC architectures are specialized instructions for handling string operations and complex memory addressing modes.

The key idea of CISC architectures is to improve program performance by reducing the overall latency of groups of instructions. This approach is inherently limited. Indeed complex instructions require complex hardware, so the critical path in their implementations represents a limit – more complex instructions requires longer critical paths.

For this reason, starting from the mid '70, computer architects focused on improving the *throughput* of completed instructions – the *Instructions Per Cycle*. An improvement of the IPC directly influences the improvements of a section of code. Indeed, the *Cycles Per Instruction*, $CPI = 1/IPC$ is inversely proportional to the instruction completion throughput.

To achieve this goal *Reduced Instruction Set Computing* architectures split the execution of a single instruction into stages. Stages are connected through a pipeline. Instructions are issued by the first stage and committed by the last. At each clock cycle, the architecture executes all stages of the pipeline in parallel, thus the clock cycle is determined by the latency of the slowest stage. The latency of an instruction is the latency of the pipeline, but instruction throughput is increased because an instruction is completed at every clock cycle.

In this schema, RISC architectures exploit parallelism between instructions in a single execution flow – *Instruction Level Parallelism* – to execute them in parallel, while still exposing a sequential programming model. Semiconductor technology improvements can be effectively exploited by RISC architectures. Indeed, new technologies allow increasing the clock frequency, thus decreasing the CPI.

RISC architectures can guarantee performance as long as they can keep the pipeline busy, that is at every clock cycle an instruction must be issued to the pipeline. If this is not possible, a bubble is inserted into the pipeline, and the overall efficiency is decreased. This behaviour is due to *hazards* induced by the instruction stream. A *control hazard* is generated when the pipeline stalls because the address of the next instruction is not yet ready. A *data hazard* arises when data needed by an instruction is not available. A *structural hazard* is due to the lack of hardware resources for executing the instruction.

In order to keep the pipeline busy, aggressive optimization techniques

```
1  cld
2  movl  $src,%esi
3  movl  $dst,%edi
4  movl  $0x400,%ecx
5  rep
6  movsb
```

(a) X86 string copy

```
1   mov r8,#0x400
2 loop_header:
3   sub   r8,r8,#0x1
4   ldrb  r9,[#src,r8]
5   strb  r9,[#dst,r8]
6   cbnz  r8,loop_header
```

(b) ARM string copy

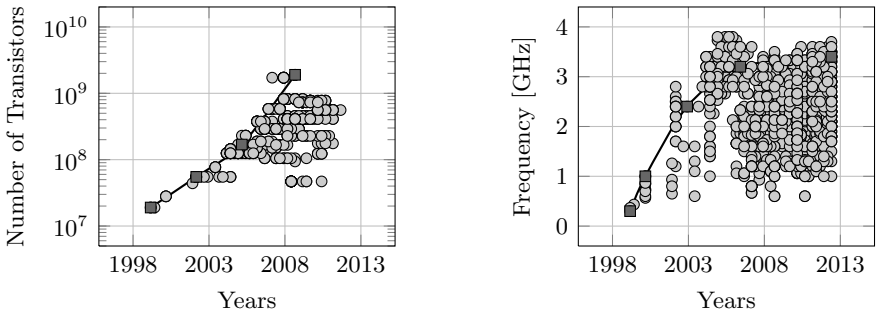
Figure 2.2: Trivial implementation of string copy on a CISC architecture (Figure 2.2(a)) and on a RISC architecture (Figure 2.2(b)). The CISC variant employs specialized instructions, while in the RISC case an explicit copy loop must be used

have been implemented along the years: pipeline forwarding, out-of-order execution [141], branch prediction [151], ... In order to increase performance, instructions have been split into a large number of short stages, thus allowing to boost clock frequency.

Example 2.1. Figure 2.2(a) shows the implementation of a 1024-elements string copy in a CISC architecture, the Intel x86 [80]. First, the direction flag is reset (Line 1) so that subsequents instructions assume strings are visited in ascending order, from the first character to the last. Then, the base pointers of the two arrays are loaded into registers `%esi` (Line 2) and `%edi` (Line 3), while, the number of elements to copy is written into register `%ecx` (Line 4). The `movsb` instruction (Line 6) copies the content of the memory cell referenced by `%esi` into the memory pointed by `%edi` , then it increments the two registers. The `rep` prefix (Line 5) instructs the hardware to execute `movsb` and to decrement the content of register `%ecx` until it reaches 0, thus it allows to copy the whole `$src` array into the `$dst` array.

Example 2.2. Figure 2.2(b) reports the implementation of a 1024-elements string copy in a RISC architecture, the ARM [17]. The instruction set is not rich as in a CISC architecture, thus an explicit loop with 1024 iterations is needed in order to copy each element from the `#src` array into the `#dst` array. The first instruction of the loop body decrements the induction variable, stored inside register `r8` (Line 3). Then, in order to copy one element, it is necessary to load it from memory to a temporary register (Line 4), and to perform a store (Line 5). Finally, a `cbnz` is executed in order to check whether there are no more iterations to execute (Line 6).

RISC architecture concepts was the driving force of computer architecture evolution up to year 2005. They was also integrated into popular



(a) Number of transistors vs years

(b) Core frequency vs years

Figure 2.3: Transistor integration. Figure 2.3(a) reports the variation of the number of transistors along the years. Reducing transistor size allows to boost the switching frequency, and thus the core operating frequency. This behaviour becomes unsustainable past year 2005 (Figure 2.3(b)), where power requirements becomes too large. Data from Intel [3]

CISC design – internally an Intel x86 core is a RISC architecture. At that time, limits of ILP [147] becomes unsurmountable walls. Mechanism needed to keep the pipeline busy require a lot of power. Moreover, increasing the clock frequency directly increases the dynamic power consumed by the architecture. This problem is generally identified with the term *power wall* [30].

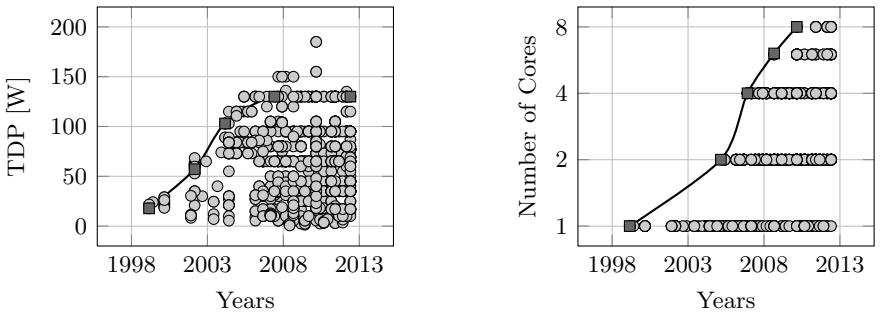
Apart of the power requirements, another problem of aggressive RISC architectures is that they primarily focus on the processor. Data hazards due to memory accesses are indirectly tackled through cache hierarchies [1], but the problem of feeding the processor with data still persists. The inability of the memory hierarchy to fulfill the data requests by the processor is identified with the term *memory wall* [150].

2.2.3 Dealing with the Power Wall

Figure 2.3(a) plots the number of transistors employed by Intel processors from 1999 to 2012 [3]. It is clear that the Moore's law is still in effect, indeed the number of transistors per die continually increases.

Looking at Figure 2.3(b) we can observe that the increased transistor density has been primarily exploited to boost the core operating frequency. However, as discussed before, increasing the switching frequency of a transistor directly influences the amount of required dynamic power. For CMOS technology, it is ruled by the following relation:

[1] Another power-hungry component



(a) TDP vs years

(b) Number of cores vs years

Figure 2.4: General purpose processors evolution. Increasing frequency to improve performance leads to power-hungry designs (Figure 2.4(a)). Acting on the number of available cores (Figure 2.4(b)) allows to contains power requirements. Data from Intel [3]

$$P_{dyn} \sim \frac{1}{2} \times C \times V^2 \times f$$

where C is the driven capacitive load, V is the working voltage and f is the switching frequency. Since C is a function of the number of transistors connected to the output and of the technology, the tunable parameters are V and f. In order to limit the consumed power while, at the same time, increasing the switching frequency, the working voltage has been progressively dropped from $5V$ to just under $1V$.

Past year 2005, the frequency of the cores did not further increase, due to the inability of exploiting ILP and of the huge power requests. The connection between frequency and power can be easily seen in Figure 2.4(a), which plots the *Thermal Design Power* of Intel processors from 1999 to 2012 [3]. The TDP growth stops at the same time of core frequency.

To contain power requests, architecture designers removed power-hungry components required by aggressive ILP techniques. With respect to processors of the ILP-era, these new processors have shorter pipelines, a narrow instruction issue window, and static scheduling instead of dynamic scheduling. Sometimes, in-order execution is preferred to out-of-order execution.

However, relying on simpler processors do not necessarily mean that we cannot execute more complex applications. Indeed, for many duties, these processors allow to execute a wide range of applications, without any performance penalty. At the same time, simple designs allow to

better accommodate emerging performance indicators, such as power consumption.

On the other hand, for some applications raw performance is an essential requisite. For instance, given a time budget, HPC applications need faster processors in order to perform more accurate simulations. Another example is related to graphic-intense applications, who aim at improving the user experience quality year after year.

To provide raw performance for such kind of applications, mechanism other than ILP should be exploited. In the general purpose CPUs market, past 1995 architecture designers started the trend of exposing parallel features to the programmer. For example, the concept of *vector* instructions has been borrowed from vector processors [127] and adapted to general purpose CPUs, leading to vector instruction set extensions [50, 118].

The exposure of hardware parallel feature to the programmer reached a critical point around 2005. Indeed, vector instruction sets just introduced new data types and instructions, thus available parallelism can be efficiently masked by well-written libraries. In order to get more performance, parallelism must be exploited in a more explicit way.

Initially, *Symmetric Multi Threading* (e.g. Intel Hyper-treading [101]) designs allowed to execute more than one independent execution flow on the same core. Stalls due to hazards in one execution flow are used to execute another execution flow. The natural evolution of this approach is *Symmetric Multi Processing*, where the architecture exposes multiple independent processing elements. This technique, initially exploited at multiple package level – i.e. installing more than one single-core processor on the same motherboard –, has been widely applied at the single package level – i.e. putting more than one core on the same die – starting from 2005.

Figure 2.4(b) reports the number of cores per Intel processors starting from 1999 up to 2012 [3]. Comparing it with Figure 2.3(a), it is clear that past 2005, increasing the number of cores became the primary way to exploit available transistors.

The modus operandi of architecture designers is clear. First, build a simple architecture to contain power requirements, then replicate the design multiple times to guarantee performance. This methodology leads to architectures that can be classified as MIMD in Flynn's taxonomy.

2.2.4 Dealing with the Memory Wall

The problem of feeding a processor with data is orthogonal to the power problem, however the same techniques used to cope with the power wall

can also be useful for fighting the memory wall.

On SISD machines there is an unique path for accessing the main memory – all accesses must pass through the *memory controller*. It is on charge of serializing all memory accesses. The primary measure of its efficiency is the *bandwidth*, that is the number of bytes it can transfer from/to the memory per second.

Along the years, the memory controller has been integrated with the processor, and they start sharing the same die. It works closely with the cache hierarchy, and, together, they are responsible for ensuring *memory consistency* [12].

Vectorial instruction set extensions usually provide instructions accessing memory with a relaxed consistency. This allows super-scalar out-of-order processors to perform a more aggressive instruction reordering and to skip the cache hierarchy while accessing memory needed for that kind of instructions. The net result is an increased bandwidth for vector-related instructions.

Due to memory consistency and caching constraints, memory accesses generated by non-vector instructions cannot use this fast-path to the memory. Considering that the memory access latency and bandwidth do not evolve like the performance of the core, it is clear that the memory controller quickly becomes a bottleneck. Moreover, past 2005, the increasing number of cores per die imposes a further load on the memory controller. Indeed, instead of feeding one core, the memory controller now has to provide data for all cores in the die.

Regardless of the number of cores, architectures where the memory access latency is constant among all processors are called *Uniform Memory Access* architectures. Figure 2.5 reports an example of a 4-core UMA architecture. Each core has a private cache hierarchy, thus as long as memory requests can be fulfilled by caches, each core can perform accesses independently.

When an access to the main memory is generated, regardless of the originating core, it must be handled by the memory controller. This is usually implemented using a simple design, such as a shared bus.

To remove the bottleneck, the number of paths to access the memory must be incremented. That is, each core must be equipped with a private memory controller, directly connected to a different memory module. Accesses to that module are called *local accesses*. When a core has to access to a memory module different from the local one, it performs a *remote access*. In that case, the access must be performed by mean of the memory controller of another core, thus with respect to a local access, the latency is greater. On the other hand, the overall bandwidth is increased.

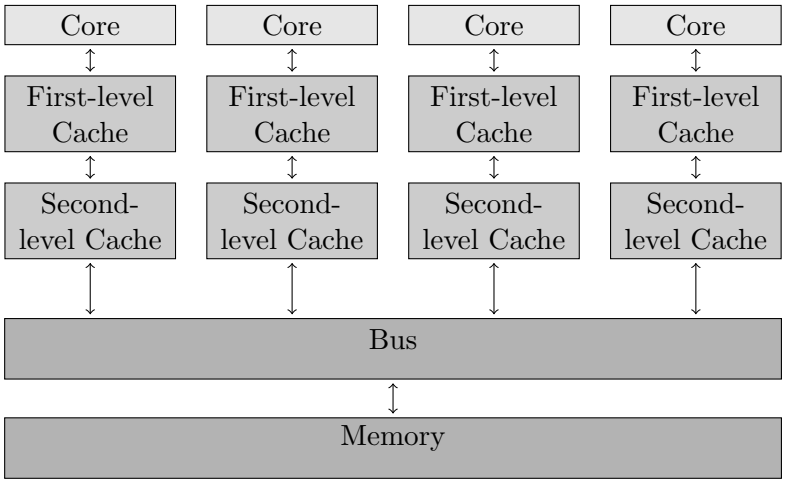


Figure 2.5: An example of UMA architecture. The latency of an access to main memory is constant across all cores. This is due to the availability of an unique path for accessing the main memory

Such kind of architectures, where the memory access latency depends on the core originating the access and on the accessed memory module are called *Non Uniform Memory Access* architectures. The core, the cache hierarchy, the memory controller, and the local memory module constitute a *node*. Communication between nodes is ensured by an *interconnect network*.

Figure 2.6 reports an example of NUMA architecture composed by four nodes, each composed by one core with a two-level private cache hierarchy.

UMA designs are still used in small multi-core architectures. When the number of available cores becomes greater than eight, NUMA designs becomes more attractive. Usually, a hybrid solution is adopted. For instance, consider an architecture composed by four interconnected processors. Each processor is equipped with four cores and is attached to a local memory module. Such an architecture has NUMA characteristics between the four processors, and UMA characteristics inside the single processor.

2.2.5 The Case of GPGPUs-based Architectures

Up to the end of the last millennium, graphics boards were in charge of handling output to the screen. The actual computation of what to display was performed on the CPU, with little assistance from the graphics

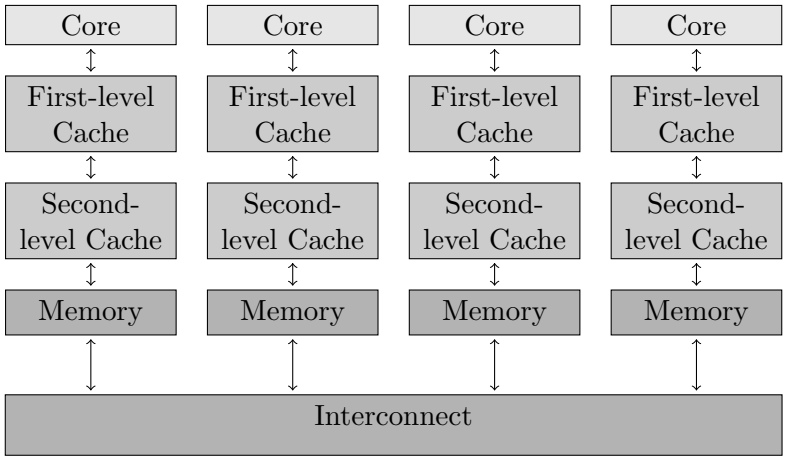


Figure 2.6: An example of NUMA architecture. The latency of an access to main memory is a function of the core where the access occurs and of the physical location of the accessed address. This is due to the need to walk the interconnect network when accessing a memory location not stored in the memory element directly connected to the core where the access initiates

board itself.

Starting with NVIDIA GeForce 265 [8], part of graphics rendering started to be offloaded on the graphics board. Graphics hardware then took the characteristics of stand-alone processors, able to substitute the CPU in performing complex graphics operations. The term *Graphics Processing Unit* was introduced to refer to that kind of graphics processors.

GPU architectures focus on performing large numbers of mostly independent operations on points – vertices – composing a scene to be rendered. This kind of computation sports remarkable similarities with the kind of scientific code known as *massively parallel*. For instance consider N-body simulations or PDE solvers. The same function is applied to all points of the input domain, to iteratively update some property – e.g. the position in the case of an astronomical N-body simulation, or the temperature in the case of a PDE solver simulating heat conduction. Functions are applied in parallel to all points in the domain, and synchronization happens only at the bounds of the computation, that is at the start/end of the iterative step.

In order to exploits GPUs for efficiently executing this kind of applications, GPU designers started supporting a limited form of programmability, leading to *General Purpose GPUs*.

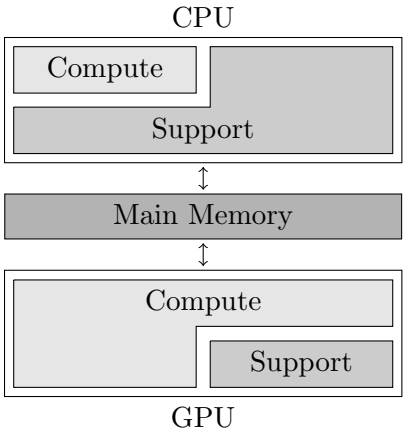


Figure 2.7: Heterogeneous computing with GPGPUs. Hot spots of the application run on the GPGPU to exploit its better computational capabilities

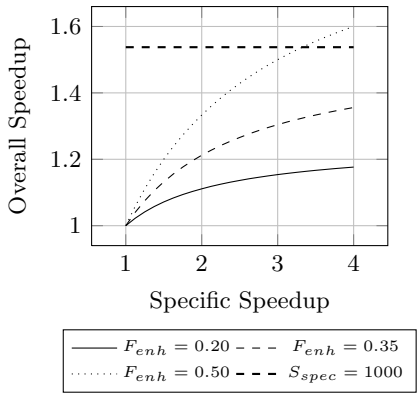


Figure 2.8: Plot of the Amdhal's law for different values of the enhanced fraction F_{enh}. The bold dashed line is the overall speedup achieved with $F_{enh} = 0.35$ and a specific speedup $S_{spec} = 1000$

GPGPUs are basically large sets of synchronous processors. They are specialized on executing massively parallel programs, with little synchronization and a regular behaviour. According to Flynn's taxonomy, GPGPUs fall into the SIMD category, although NVIDIA refers to them using the *Single Instruction Multiple Threads* name, pointing out the ability of executing SIMD code with no fixed vector width [7], thus with an increased efficiency with respect to the most widespread SIMD architecture at that time – vectorial instruction set extension.

Starting from NVIDIA GeForce8800GT [9], massively parallel processors start to be available at modest prices. This lead to a renewed interest in heterogeneous architectures, that could be composed by a general purpose processor, and one or more GPGPUs.

In the GPGPUs setting, the general purpose processor is used to drive the computation. If application hot spots exhibit regular behaviours, they can be offloaded to the GPGPUs, taking advantage of the more specialized hardware. The control processor and the accelerator communicates through the main memory. An example of this setup is depicted in Figure 2.7.

With respect to a typical CPU, a GPGPU dedicates a higher ratio of its area to computational resources. This allows excellent speedups [142] on applications which hot spots can be efficiently mapped on the GPGPUs. On the other hand, the greater die area dedicated to support

computation on the CPU – e.g. caches – allows to execute a more wider range of applications with reasonable efficiency.

There is a growing trend in the semiconductor industry towards designs that take cues from the GPGPU experience – e.g. coupling CPU cores with heterogeneous parallel accelerator on the same die [2, 152]. The goal is to minimize the overhead of performing data movement between the CPU and the GPU. On the other hand, in HPC environments, stand-alone architectures still dominates the market [4, 10], due to the request of higher peak performance.

2.2.6 Taking into Account the Amdahl's Law

Given a program and an optimization, let F_{enh} the fraction of the program improved by the optimization and S_{spec} the speedup due to the optimization on the fraction F_{enh}. Amdahl's law states that the overall speedup of the program due to the optimization S_{over} is related to the enhanced fraction F_{enh}:

$$S_{over} = \frac{1}{(1 - F_{enh}) + \frac{F_{enh}}{S_{spec}}}$$

The main consequence of Amdahl's law is that the maximum achievable speedup of an optimization is bound by the optimized fraction:

$$\lim_{S_{spec} \to +\infty} S_{over} = \frac{1}{1 - F_{enh}}$$

Figure 2.8 depicts this behaviour. It plots the overall speedup S_{over} for different values of the enhanced fraction F_{enh} varying the specific speedup S_{spec}. The dashed bold line represents the overall speedup that can be achieved with a specific speedup of 1000x over the 30% of a program. Comparing it with the dotted line, representing the overall speedup when optimizing 50% of the program, one can observe that a more modest speedup of 4x is more effective.

Performance gains expected due to parallel execution are also subject to Amdahl's law. For instance, consider the case where a fraction of the program can be perfectly parallelized. The specific speedup due to the optimization S_{spec} is thus linear in the number of available processing elements N_{pe}. The Amdahl's law can be rewritten as following:

$$S_{over} = \frac{1}{(1 - F_{enh}) + \frac{F_{enh}}{N_{pe}}}$$

It does not matter how many processing elements are available in the architecture. The overall speedup is still bounded by the unoptimized

section of the program, in this case the *sequential part*, thus increasing the number of processing elements does not guarantee performance scaling.

2.3 Parallel Programming Models

Explicitly exposing hardware parallel features allows architecture designers to increase the overall hardware performance. On the other hand, with respect to past advancements, this kind of greater performance improvement comes with greater responsibility [98].

To take advantage of parallel processing elements, two different approaches can be followed: *implicit* and *explicit* techniques.

The goal of implicit techniques is to hide as much as possible the parallelism available in the target architecture. Actual exploitation of parallelism is either achieved at run-time or at compile-time. For instance consider ILP: the hardware dynamically detects independent instructions in programs, executing them in parallel using different functional units.

As stated before, ILP cannot guarantee further performance improvements. Automatic parallelization techniques try to preserve the abstraction of sequential programming model, by exploiting compile-time techniques to split programs into independent parts. However, their effectiveness is limited by the structure of the program: techniques based on the analysis of loop indexes, such as *Polyhedral Analysis* [13], requires loops with a fairly regular structure, while attempts to auto-parallelize general-purpose code – e.g. *Decoupled SoftWare Pipelining* [114] – have to deal with inabilities of compilers to provide accurate analysis of program properties, such as alias sets.

In order to fully exploit parallel processing elements, explicit techniques rely on exposing the hardware parallelism to programmers. This allows to communicate more information to the compiler, that can in turn generate more efficient code. Furthermore, with respect to automatic parallelization techniques, programmers can identify larger parallel sections, thus potentially achieving better speedups – we must shift from sequential to parallel programming models.

2.3.1 Programming Parallel Architectures

A programming model is an abstraction of an architecture, which models how computations are described and run. A programming language implements a programming model, thus allowing to write programs according to it.

The programming model concept has a strong connection with the hardware architecture. Indeed, almost all architectures can be programmed using different programming models, but, usually, given an architecture, there is one programming model that allows to fully exploits its resources. For instance, programs written in an imperative way can fully exploit a general purpose single-core processor.

When dealing with programming models – and architectures – built around the concept of a single execution flow, writing a program usually is just one step of a relatively simple process. Given a problem, first an algorithm that solves it must be found. Then, a program implementing the algorithm is written. Finally, the program is compiled and run on the target architecture.

In this scenario, we can increase the performance of the program in different ways: improve the algorithm, tune the implementation, or buy better hardware. These three actions are usually quite independent.

Parallel programming models are built around the concept of multiple execution flows. They are usually specialized in handling some kind of computations and/or are suited for certain class of parallel architectures.

The aforementioned process of writing and optimizing programs cannot be used with parallel programming models. Indeed, the algorithm, the program, and the architecture are all closely related. Working on one component at time does not guarantee performance.

For instance consider the problem of sorting data. If we are going to target a GPGPUs, employing a comparison-based sorting algorithm, like quick-sort, does not guarantee efficient exploitation of hardware resources. On the other hand, bitonic-sort perfectly matches GPGPU hardware features, so it is the ideal candidate. Finally, we must take into account the program, thus the algorithm must be coded according to GPGPUs best practices – e.g. correct handling of memory hierarchy and coalesced accesses.

2.3.2 Data-parallel Programming Models

One of the first parallel programming models arises from the needs of the High Performance Computing community. Typical applications include computational fluid dynamics, molecular dynamics, and astrophysics simulations. A key parameter is the *problem size*, that is an indicator of the dimension of the problem to be solved. It is often related to the size of input data or to the accuracy of simulations.

To cope with increasingly larger problems, some kind of parallelism is necessary. A key aspect of HPC applications is that many operations are *naturally parallel*. Figure 2.9 shows one example: Single precisions A

```
1 void saxpy(int n, float a, float *x, float *y) {
2   for(int i = 0; i < n; ++i)
3     y[i] += a * x[i];
4 }
```

Figure 2.9: The SAXPY kernel is a BLAS level-1 routine. From the programming model point of view, its key feature is the absence of dependencies between different iterations of the loop

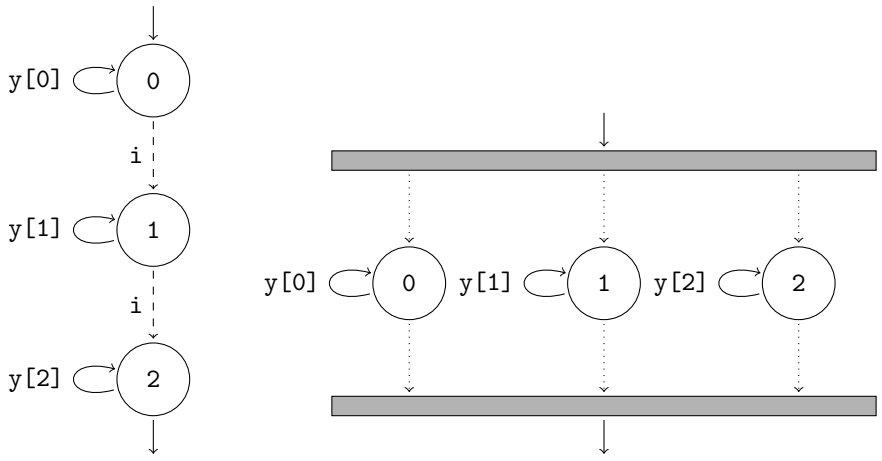
times X Plus Y, a level-1 BLAS routine [94]. Given two arrays of floating point values `y` and `x`, and a floating point scalar `a`, every element of the `y` array is incremented by the corresponding element of array `x` times `a`.

The update of a single element of the array is a simple operation, and cannot be parallelized. On the other hand, if we consider two different updates, we can see that they are completely independent. The cause of greater performance needs – the large working set – is in this case also the mean to achieve greater performance.

Data-parallel programming models target this kind of applications. It is often said that an application suitable for a data-parallel programming model exhibits a *regular behaviour*: it is not control-intensive and it performs a large number of operations, often on large arrays accessed with fixed strides.

Figure 2.10(a) reports the sequential execution of a SAXPY kernel, together with its data dependencies. Each node represents an iteration of the loop. An arc between two nodes q and r represents a dependency. Solid arcs represent dependencies that cannot be eliminated. For example, in each iteration a cell of array `y` is read and later written (WAR dependency) – there is no way to get rid of this dependency. Dashed arcs represent dependencies that can be eliminated. For instance, the loop induction variable `i` is only used to detect the array elements accessed by each iteration. It has a regular behaviour: during each iteration it is read and then incremented by a constant amount. This allows to predict the value of `i` in each iteration, so the associated loop-carried dependency can be eliminated. The dependency graphs becomes unconnected, and thus all the unconnected components – the iterations – can be executed in parallel.

If the single iteration is composed by a small number of simple instructions, vector processors or vector instruction set extensions allows to mask parallel iterations execution behind vector data types – the loop is said to be *vectorized*. Automatic compiler-based parallelization techniques, such as the ones based on polyhedral analysis [13], are also



(a) Sequential (b) Parallel

Figure 2.10: Sequential and parallel execution of SAXPY. A node marked with label `i` represents the `i`-th iteration of the loop. The only loop-carried dependency is the one related to the loop induction variable `i` (Figure 2.10(a)), thus all iterations can potentially be executed in parallel (Figure 2.10(b))

effective in parallelizing such simple loops.

When the complexity of the iteration increases, these techniques cannot be used. Indeed, instructions composing the iteration cannot be mapped to vector instructions, and automatic techniques fails in precisely detecting all dependencies related to the iteration – they cannot parallelize the loop.

For instance, automatic-parallelization techniques requires a perfect knowledge of both loop structure and dependencies of the loop. Without these information, they cannot proceed with automatic parallelization, because they cannot guarantee that the original semantic of the program is preserved.

The Fork-Join Programming Abstraction

Another approach to exploiting data-parallelism is the concept of explicit *parallel loop*. All of its iterations are guaranteed to be independent, so they can be executed in any order, and therefore in parallel. The reference model is known as *fork-join*.

Independent execution flows are identified with the term *thread*. When a thread reaches a parallel loop, it spawns a parallel computation, and

thus it creates – *forks* – a certain number of threads to help him executing the iterations of the parallel loop. The spawning thread is called *master thread*, while the spawned threads are called *slave threads*. Iterations of the loop are thus distributed across all threads, and executed in parallel. When there are no more iterations to execute, all threads meet at the end of the parallel loop – with a *join* operation. Spawned threads terminate their execution, while the master thread continues executing with the statement following the parallel loop.

Figure 2.10(b) shows the execution of SAXPY using data-parallelism. Parallel execution is managed according to the fork-join model. The original SAXPY loop is considered a parallel loop, and thus all iterations can be executed in parallel. The top gray rectangle represents the fork phase, while the one at the bottom represents the join phase.

Parallel loops are the basic building block of data-parallel programming models. In all of them, the programmer must assist the compiler and/or the runtime to detect parallel loops.

In Fortran it is not possible to define a parallel loop, but several statements allows to express natural parallel computations. Indeed it is possible to perform member-wise operations between arrays or matrices. The Fortran compiler can take advantage of this information to either generate vector code or parallel code.

The OpenMP programming model [16] provides a set of compiler directives allowing to tag parallel loops. It is an extension of C, C++, and Fortran languages. Directives are exploited to partition iterations either at compile-time, by evenly dividing iterations across threads, or at run-time, to better adapt to unbalanced workloads. Threads are usually executed by general purpose multi-core processors, but some attempts have been made to target other architectures, such as GPGPUs [97]. Data is accessed by threads through shared memory.

Figure 2.11 reports the SAXPY kernel parallelized thought OpenMP directives. The `#pragma omp parallel for` directive allows to declare the tagged loop as a parallel one. The compiler emits the code of both master and slave threads. Just before entering the parallel loop, the master thread evenly divides iterations across available slave threads. Usually, the OpenMP runtime use all cores of the target machine to execute the parallel loop. Let N_{pe} the number of cores, since one core is used by the master thread, $N_{pe} - 1$ slave threads are created. Together with the master thread, they execute the n iterations of the parallel loop, so each thread executes at most n/N_{pe} iterations.

In the OpenMP programming model there is a single unified address space, shared among all threads. While this facilitates programs writing, it constitutes a problem when targeting NUMA architectures. In-

```
1 void saxpy(int n, float a, float *x, float *y) {
2   int i;
3
4   #pragma omp parallel for
5   for(i = 0; i < n; ++i)
6     y[i] += a * x[i];
7 }
```

Figure 2.11: OpenMP version of the SAXPY kernel. The pragma attached to the loop is an hint to the compiler and to the runtime. It states that all loop iterations can be executed in parallel

deed, equally partitioning parallel loop iterations among available worker threads can generate an uneven partition, due to the different latency of memory accesses.

Dealing with NUMA Effects

In the context of shared memory programming models, *Partitioned Global Address Space* languages divide the shared address space into sections. Data structures are manually distributed by programmers on those sections. Distribution allows to take advantage of NUMA features on underlying architecture. Parallel computations are mainly expressed by means of parallel loops, together with locality constraints – i.e. programmers must specify which iterations must be executed by each thread. Unified Parallel C [143], Fortress [14], Chapel [44], and X10 [128] are the best known PGAS languages.

Massively Parallel Languages

According to Amdahl's law, optimizations must insist on application hot spots. The effectiveness of an optimization is both related to its specific performance gain and to its applicability. From the architecture point of view, an heterogeneous design, composed by multiple processing elements, each optimized for executing a particular kind of code, allows to boost the specific speedup.

The duty of a programming model for an heterogeneous architecture is to allow efficient exploitation of high specific speedups through language constructs that should be used to express hot spot computations.

In the context of the data-parallel programming model, CUDA [6] and OpenCL [87] both focus on heterogeneous architectures. The two proposals are quite similar, so the following considerations about OpenCL

can also be applied to CUDA.

Programs are organized into an *host part* and a *device part*. The host part usually run on a general purpose processor, and it is in charge of controlling heavyweight computations spawned on the device. These computations are application hotspots. In the OpenCL terminology, they are called *N-Dimensional Range kernels*, but it fact each of them represents a perfect loop nest. Loop iterations are called *work-items*. They can be grouped according to a hyper-rectangle [2] geometry into *work-groups*.

With respect to OpenMP, the master thread role is taken by the thread running on the host. It simply submits NDRange kernels to the device in form of a set of homogeneous work-groups. The device executes them, possibly exploiting two level of parallelism: between work-groups and between work-items. Indeed work-groups are completely independent, and each of them is composed by independent work-items. Synchronization can be achieved only inside work-groups – i.e. work-items belonging to a work-group can employ barrier constructs to meet at a synchronization point.

The term *massively parallel* is often associated with both CUDA and OpenCL. It refers to the huge number of available processing elements in target architectures – usually a GPGPU –, and to the features of applications suited for these languages, which usually expose a large parallelism degree, with little or no synchronization between work-items.

CUDA and OpenCL both aim at achieving the maximal exploitation of computing resources. To do this, a programmer is given complete control of the target device. This approach, however, requires writing a lot of boilerplate code, such as transferring data to the accelerator. Moreover, performances are very sensible to even minimal variation in the architectural parameters. To cope with these problems, OpenMP-like extensions to C, C++, and Fortran have been proposed [11, 113], but they cannot compete with the performance of hand-written code.

Aggregating Values

An important class of computations involved in the process of large data-sets is the *reduction*. It consists of producing an aggregate value out of a set of values. In the context of HPC, scalar product is a good representative of this kind of computation. Figure 2.12 shows a serial implementation of the SDOT BLAS level-1 routine [94].

This kind of computation represents a challenge for every data-parallel programming model. Problems arise from the dependency in the SDOT

[2] CNN thinks orthotope is better

```
1 float sdot(int n, float *x, float *y) {
2   float sum = 0.0;
3
4   for(int i = 0; i < n; ++i)
5     sum += x[i] * y[i];
6
7   return sum;
8 }
```

Figure 2.12: The SDOT kernel is a BLAS level-1 routine. The `sum` variable is used to accumulate the partial value of the dot product. Since it is updated at every loop iteration, a loop-carried dependency is induced

loop. While there are no dependencies on the accessed arrays, there is a loop-carried dependency related to the update of the accumulator used during the reduction – the `sum` variable. In Figure 2.13(a) the loop-carried dependency is represented by a solid arc between each pair of iterations.

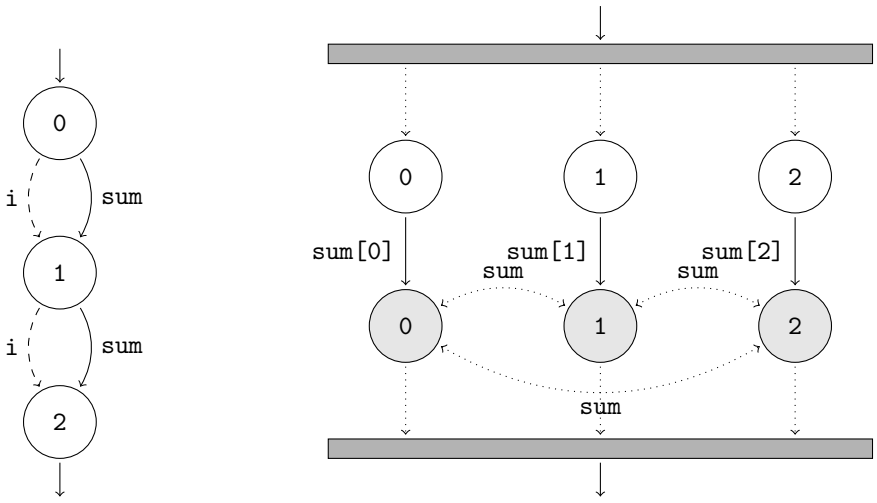
To parallelize the loop, the dependency on `sum` must be eliminated. Since there are no other dependencies in the loop, the key idea is to execute all iterations in parallel. Each iteration first compute a local reduction, then it executes a communication step, in order to aggregate its partial reduction value to the global one. Figure 2.13(b) depicts this process, where light gray nodes represent communication steps.

With respect to the SAXPY example, reported in Figure 2.10, parallelizing SDOT requires restructuring the body of the loop, to break the loop-carried dependency. However, during the split we have assumed the order in which `sum` is updated does not matter, that is `sum` is updated by means of an associative and commutative operator.

This assumption does not always hold. For example, consider a loop were each iteration appends a character to a string. Since the appending operation is neither associative nor commutative, we cannot parallelize the loop in this way.

Data-parallel programming models allow to perform reductions on variables involving associative and commutative operators. Usually, a directive can be attached to a parallel loop, to inform the compiler about both the accumulator, and the operator to be used for it. Figure 2.14 reports an OpenMP version of SDOT. The `reduction` clause identifies reduction-related information.

Generally speaking, operators and data-types involved in reductions are defined by language specifications. However some languages [14, 44]



(a) Sequential (b) Parallel

Figure 2.13: Sequential and parallel execution of SDOT. A node marked with label i-th represents the i-th iteration of the loop. Parallel execution (Figure 2.13(b)) is achieved by eliminating the sum loop-carried dependency (Figure 2.13(a)) through privatization, together with a communication step at the end of each iteration – light gray nodes

```
1 float sdot(int n, float *x, float *y) {
2   float sum = 0.0;
3   int i;
4
5   #pragma omp parallel for reduction(+:sum)
6   for(i = 0; i < n; ++i)
7     sum += x[i] * y[i];
8
9   return sum;
10 }
```

Figure 2.14: SDOT kernel parallelized via OpenMP directives. The `reduction` clause informs the compiler to privatize `sum`, and to aggregate partial reductions using the `+` operator

allow using user-defined reductions, involving more complex operations and/or data-types.

CUDA and OpenCL are a special case. There is no built-in support for reductions, but to fully exploit the target architecture some coding patterns – e.g. hierarchical reductions – must be exploited.

From a performance point of view, reductions are critical operations because they induce a bottleneck in parallel execution. Indeed, accesses to the shared accumulator must be coordinated during updates. This requires serializing the execution, and thus can limit the scalability of the application.

2.3.3 Task-parallel Programming Models

The absence of regularities in the behaviour of an application does not prevent its parallelization. It simply means that the data-parallel programming model is not suitable. For example, consider the problem of recursively ordering an array. It is clearly control-intensive, and the data access pattern is not ruled by the algorithm, but by the values found in the array at run-time.

Recursive ordering can be achieved through a simple scheme: split the array into two sections, recursively order them, and then merge the two ordered partitions. The recursion steps operate on disjoint portions of the array, so they can be executed in parallel due to the absence of any dependency. On the other hand, merging must be executed after the two parallel sorting terminates.

Most of the irregularity of this problem comes from the fact that the number of potentially parallel recursive calls is not known. A compiler cannot predict, even symbolically, this number, because of dependencies on run-time values. At run-time, just before starting the sort operation, it is also not possible to predict the number of recursive calls, because the computation will be ruled by data found in the array. Parallelism is discovered while executing the application, so a programming model able to generate parallel computations at run-time is needed.

Task-parallel programming models are built around the abstraction of *task*, that is a chunk of parallel computation, usually identified by tagging a section of code or a function. Task definition only identifies the unit of parallelism. To start a parallel computation, the programmer explicitly creates new tasks. This is referred to with the term *spawning*. By spawning a task, the programmer simply declares that it can be executed in parallel with the current execution flow. It is up to the runtime to execute the task at the most suitable time. Synchronization between tasks is usually achieved by *joining*: a task can wait for the

termination of another task before proceeding with its execution.

Relationships between tasks can be represented by means of a *task-graph*. Each node in the graph represents a spawned task. There are two kind of arcs: spawning and joining. A spawning arc from a node q to a node r, means that task q has spawned task r, while a joining arc from a node q to a node r means that task r has joined task q – task r has waited for q completion.

The best known implementation of the task-based parallel programming model is Cilk [56]. It is based on the C language. Tasks are functions tagged with the `cilk` keyword. To spawn a task, the programmer invokes a task by preceding it with the `spawn` keyword. A task can wait for the termination of all spawned tasks using the `sync` statement. Before terminating its execution, a task must wait for the termination of all tasks it has spawned – an implicit `sync` statement is executed at the end of each task.

Figure 2.15 shows the implementation of recursive array ordering in Cilk. The `cilk_sort` function represents a task. Input parameters are the array to order `x`, a support array `tmp`, and the length of both arrays `n`. The base case is represented by sorting a "small" array, which is performed through quick sort (Line 3). For bigger arrays, `cilk_sort` call itself recursively twice (Line 7 and Line 8). This spawns a couple of tasks, each ordering a section of the input array `x`. Before proceeding with the merge step (Line 12), each tasks needs to wait for the termination of spawned tasks (Line 10).

A strong property of the data-parallel programming model is that the structure of the parallel computation is known; it behaves according to the fork-join model. This allows to perform part of the scheduling decisions at compile-time, thus reducing the run-time overhead.

With the task-parallel programming model, this is not possible. Indeed, while the ability of spawning tasks on-demand, allows to take advantage of all parallelism discovered at run-time, it ends up in generating arbitrary task-graphs, with no regularities. No scheduling decisions can be performed at compile-time, and thus higher run-time overhead are expected.

The problem can be partially addressed, by enforcing tasks to be spawned and joined with a "nice" behaviour. In the case of Cilk, the implicit `sync` statement at the end of each task induces a particular organization of the task-graph. All computations are said to be *fully strict*, and aggressive run-time scheduling techniques can be applied [29].

Moreover, the ability of spawning tasks must walk together with the programmer responsibility of identifying when to stop spawning them. Indeed, past this cut-off value, the overhead of spawning parallel com-

```
1 cilk void cilk_sort(float *x, float *tmp, int n) {
2   if(n < QUICK_SIZE) {
3     quick_sort(x, n);
4     return;
5   }
6
7   spawn cilk_sort(x, tmp, n / 2);
8   spawn cilk_sort(x + n / 2, tmp + n / 2, n / 2);
9
10  sync;
11
12  spawn cilk_merge(tmp, tmp + n / 2, x);
13 }
```

Figure 2.15: Parallel sorting in Cilk. The array `x` is ordered following a divide-et-impera approach. Recursive calls to `cilk_sort` can be executed in parallel – they work on different sections of the `x` array. The `sync` statement ensures that `cilk_merge` is called after the two recursive sorts complete

putations is too big, and it is not balanced by the expected performance gain.

The task-based parallel programming model is implemented by many languages and libraries. Among the others, X10 [128] handles tasks according to a relaxed version of Cilk fully strict computations [66], while in the Intel Thread Building Blocks C++ library [5], the task is a central concept – it is also used to execute data-parallel computations. OpenCL [87] allows to execute tasks, but this is equivalent to executing kernels consisting of just one work-group made up by one work-item. Moreover, it is not possible spawning new tasks inside a task. On the other hand, through the usage of out-of-order queues, it is possible exploiting a limited version of task-parallelism.

2.3.4 Data-flow Parallel Programming Models

In many programming models, data assumes a passive role. Even where there is a strong attention to data (e.g. the object-oriented programming model), it does not collaborate on defining the structure of the application. At most, it influences the choice of data-structures and/or algorithms employed by a particular application.

However, some applications can be modelled according to the accessed data. For example, consider a signal-processing application, like a FM-

radio. It consists of different filters, connected through a pipeline. The first stage captures the signal from an hardware medium, while the last reproduces the signal. Central stages decode the signal.

A stage is essentially a description of an elementary operation. It reads some data samples from the previous stages, which are then transformed into new samples, and finally communicated to next stages. The whole process can be described by means of a *data-flow graph*, where nodes represent stages and arcs represent connection between couple of nodes.

Given a couple of nodes q and r, the connecting arc $q \to r$ is labelled with an input rate q_{out} and an output rate r_{in}. This models the fact that when the computation associated with state q is executed, q_{out} samples are emitted on arc $q \to r$. The output rate r_{in} is related to the activation of node r. Indeed, when node q has emitted at least r_{in} samples, the computation related to node r can be executed. Arcs act as a medium in transferring data from nodes, thus they are often identified with the term *channels*.

It is clear that data is driving the computation – when channel constraints are satisfied, computation is triggered – so, this class of programming models is called *data-flow programming model*. StreamIt [62] is the best representative implementation.

Figure 2.16 shows a simple StreamIt example: a low pass filter. Stages are identified with `filter` blocks. Each filter is characterized by an input and an output channel, described by means of a `->` operator connecting the type of elements read/written from/to channels. The `void` type has the meaning of no-channel, so as the case of `FloatSource` (Line 7), there is only one output channel, that transfers `float` values to the next stage. The `LowPassFilter` (Line 18) reads `float` values from the previous stage and write them to the next, while the `FloatPrinter` (Line 25) just consumes `float` values.

A `work` action is associated to each filter. It describes the computation performed by the filter. Moreover it declares how many elements are read from the input channel, and how many are written to the output channel for each activation of the filter. In the case of `FloatPrinter`, at each activation, one element is consumed – `pop` – from the input channel (Line 26). The `FloatSource` filter exhibits the opposite behaviour: at each activation, one element is produced – `push` – on the output channel (Line 12. Obviously, is it possible to both consume and produce elements, such as in the case of `LowPassFilter` – not reported for brevity.

Finally, each filter can have a state. For instance, the `FloatSource` filter has a state (Line 8), representing the next element to push on the output channel. The `FloatPrinter` is an example of a stateless filter – it is only made of the `work` function.

```
1  void->void pipeline Pipe {
2    add FloatSource();
3    add LowPassFilter(250000000, 1000, 64, 2);
4    add FloatPrinter();
5  }
6
7  void->float filter FloatSource {
8    float x;
9
10   init { x = 0; }
11
12   work push 1 {
13     push(x)
14     x = x + 1.0;
15   }
16 }
17
18 float->float filter LowPassFilter(float rate,
19                                    float cutOff,
20                                    int taps,
21                                    int decimation) {
22   ...
23 }
24
25 float->void filter FloatPrinter {
26   work pop 1 {
27     println(pop());
28   }
29 }
```

Figure 2.16: A StreamIt low pass filter. The filtering pipeline `Pipe` is composed by 3 stages. The `work` block associated to each filter specify the action to execute; `FloatSource` produces samples for `LowPassFilter`. The last stage, `FloatPrinter`, prints filtered values on standard output

The `pipeline` block allows to connect stages into a pipeline. In this case, the aforementioned stages are connected one after the other, in a linear fashion (Line 1), but more complex data-flow graphs, involving stream splitting, joining, and back-edges, are possible.

The data-flow parallel programming model does not focus on parallelism, it focus on data. Programmers identify sequence of operations that allow to transform an input data stream to an output data stream. Thanks to the explicit representation of communications by mean of arcs, it is possible to analyze the data-flow graph, and to build a stage-firing sequence at compile-time. Moreover, having multiple processing elements, it is possible to execute more than one stage at the same time. Indeed, the whole pipeline advances in lock-step mode, and in the steady-state, the current step of each stage can be executed in parallel [63] with the others.

Languages implementing the data-flow parallel programming model usually provide near-zero runtime overheads, due to the compile-time schedule of the data-flow graph. On the other hand, this simple model prevents the expression of all possible computations. For example, it is not possible to employ channels with variable capacity, and aggressive compile-time optimizations can be achieved only with a subset of possible data-flow graphs. However, using this kind of data-flow graphs one can model a significant number of applications [139], so this programming model has reached some degree of interest both in academy and in industry, especially in the domain of signal processing.

2.3.5 Task/Data-flow Parallel Programming Models

The task-parallel programming model allows to express arbitrary irregular applications, but does not address the problem of synchronizing data accesses performed by different tasks. Indeed, if at least two tasks can update concurrently a chunk of memory, synchronization primitives must be employed to avoid a data race.

On the other hand, the data-flow parallel programming model abstracts from how the computation is executed, focusing on data dependencies of the application. This allows the elimination of explicit synchronization statements, at the cost of a constrained programming model.

The *task/data-flow parallel programming model* tries to get the best of the two approaches. Basically, it acts like a task-parallel programming model. The programmer identifies and spawns tasks explicitly, but each task is also described by its data dependencies. Tasks are organized in a task-graph according to their data dependencies, which are tracked

at run-time. Tasks with no predecessors represent ready tasks. The runtime is responsible of moving them from the task-graph to the ready queues of worker threads that will execute them.

Data dependencies refer to memory locations, identified by means of a base address and a size. An input dependency models a location that will be read by the task, while an output dependency is related to a location that will be written by the task. An input/output dependency includes both. It is clear that input dependencies rule the execution of the task. Indeed, either the referred location has no known dependencies, or the referred location has to be written by another task – it is one of its output dependencies.

Figure 2.17 reports the implementation of Cholesky decomposition exploiting the task-data flow parallel programming model provided by the StarSs [24, 35] language. The input of function `cholesky` is the `NT` × `NT` blocked matrix `A`. Each block is a `TS` × `TS` sub-matrix. The algorithm applies different operations on blocks of matrix `A`, without any explicit synchronization. Every call to `spotrf` (Line 3), `strsm` (Line 6), `sgemm` (Line 10), and `ssyrk` (Line 12) spawns a new task, by inserting it into the task-graph according to its data dependencies.

Tasks and dependencies are identified by tagging functions or code blocks. For example, the function `spotrf` (Line 20) is a task accessing a memory chunk through the formal parameter `A`. The `inout` clause tells the StarSs compiler and runtime that the chunk referenced by `A` is an input/output dependence of size `TS` × `TS` × `sizeof(float)`. In the case of the `strsm` function (Line 24), both parameters refer to a matrix with the same size, but only `B` is an input/output dependency. Matrices referenced through parameter `T` will be accessed in read-only mode.

The master thread continues spawning tasks until the end of the function, where it waits for slave threads to finish executing all spawned tasks (Line 16). Meanwhile, as soon as a task completes, its output dependencies are declared satisfied and tasks with corresponding input dependencies can be moved to the ready queues.

For instance, at the first iteration of the outer loop, the master thread spawns an instance of task `spotrf`. Then it starts spawning `strsm` tasks. The first spawned `strsm` task, must wait for the spawned `sportf`, because of the dependence between `spotrf` parameter `A` and `strsm` parameter `T`. As the calculation progress, more dependencies will be resolved, and thus the parallelism degree of the application increases.

With respect to the task-parallel programming model, a lesser amount of manual synchronization has is required, thus a reduction of programming errors related to data-races is expected. On the other hand, the data-flow graph does not expose any regularity, so that aggressive run-

```
1  void cholesky(float **A) {
2    for(int k = 0; k < NT; ++k) {
3      spotrf(A[k*NT + k]);
4
5      for(int i = k + 1; i < NT; ++i)
6        strsm(A[k*NT + k], A[k*NT + i]);
7
8      for(int i = k + 1; i < NT; ++i) {
9        for(int j = k + 1; j < i; ++j)
10         sgemm(A[k*NT + i], A[k*NT + i], A[j*NT + i]);
11
12       ssyrk(A[k*NT + i], A[i*NT + i]);
13     }
14   }
15
16   #pragma css taskwait
17 }
18
19 #pragma css task inout([TS][TS]A)
20 void spotrf(float *A);
21
22 #pragma css task input([TS][TS]T) \
23                  inout([TS][TS]B)
24 void strsm(float *T, float *B);
25
26 #pragma css task input([TS][TS]A, [TS][TS]B) \
27                  inout([TS][TS]C)
28 void sgemm(float *A, float *B, float *C);
29
30 #pragma css task input([TS][TS]A) \
31                  inout([TS][TS]C)
32 void ssyrk(float *A, float *C);
```

Figure 2.17: Cholesky decomposition using StarSs. The `cholesky` function operates on the blocked symmetric positive-defined matrix `A`. Each block is a `TS` $\times$ `TS` elements sub-matrix, and matrix `A` is composed by `NT` $\times$ `NT` blocks. Employed routines are actually tasks, thus they can be executed in parallel with the task running `cholesky`, according to data dependencies expressed by mean of `input`, `output`, and `inout` clauses

time techniques cannot be exploited. However, it is possible combining a task-data flow scheduler with a traditional task scheduler, allowing aggressive scheduling techniques on tasks with no explicit data dependencies, as shown in [144].

2.4 Workload Analysis

The traditional way for evaluating the performance of a computer system requires running a benchmark suite and comparing the output score with a reference value. Actually, the role of a benchmark suite is to evaluate the performance of the computing system under a given workload, hence for each scenario a specific benchmark suite is needed.

In the realm of sequential computing, the features of benchmark suites are well known. Usually general purpose suites employ a mix of programs written using mainstream languages. For instance, in the case of SPEC CPU2006 [71], benchmarks are written in C, C++ and Fortran. The usage of C and C++ is due to their relevance in system software, while Fortran has been chosen due to its usage in numerical code. If we consider enterprise applications, benchmark suites, such as DaCapo [28], are often based on Java applications. In any case, a system is exercised to check whether it can efficiently execute *current applications* on architectures based on the SISD model. If multi-threaded applications are included in the suite, they often exploit an embarrassing parallelization scheme, with highly independent parallel portion, and almost zero communication cost, such as in the case of multi-threaded servers.

When looking at parallel computing, most of benchmark suites come from the HPC domain. The dominant language is Fortran, while parallelization is expressed by means of explicit parallel constructs, such as OpenMP parallel loops or MPI processes, as done in SPEC OMP2001 [19] and SPEC MPI2007 [108]. The first targets small scale parallel computers, where processing units share a common address space, while the second is specialized in benchmarking large scale installations, where coordination is achieved by means of message passing.

With respect to benchmark suites defined for sequential applications, we have to deal with more hardware configurations. Indeed, since a reference parallel architecture is not available, benchmark results cannot be easily generalized. For instance, consider an optimized algorithm, which MPI implementation achieves good performance. We cannot say that performance will be preserved if we change the size of the cluster. This enforces us to evaluate the algorithm with different hardware configurations, trying to detect how it scales, that is how efficiently it

employs available parallel processing units,

This issue suggest us that in the context of parallel computing, benchmarking should be a challenging task, especially when considering applications other than the ones related to the HPC domain. The main problem is that there is no consolidated environment that allows us to extract relevant program features inside a benchmark and to stress a well known architecture. The only thing we know is that we have to exploit parallel computing to increase performance, but we have not a wide spectrum of parallel applications from which extracting a benchmark suite, we do not known the target architecture, and we do not known which programming model to use.

Moreover, the lack of a clear benchmarking environment does not facilitate the comparison between different architectures and/or programming models, especially when considering not only the raw wall clock, but also emerging requirements such as limited power budgets, or language usability.

On the other hand, this is also an opportunity to reverse the research perspective. In the past, computer architects added new features, such as SIMD instruction set extensions, hoping that they would be useful for application programmers. Languages and programming models were extended to support these features, by exporting them to the application programmer. Now, instead of focusing on benchmarks, it is necessary to identify a set of interesting kernels for next generation applications. They should be used both to drive research in parallel computing – i.e. improving support for future applications needs instead of providing new features, hoping that they will be useful – and to establish a common baseline for benchmark definitions and comparison of non-functional features, such as the usability of a language.

Those kernels are called patterns, but they are not related to the design of applications, such as the ones in [58]. They refer to important problems, such as dense linear algebra routines or graph traversal.

Relevant patterns were initially searched in HPC applications, leading to the identification of 7 common kernels [42], called *dwarfs*, each mainly characterized by how data is accessed, and thus by the exposed communication pattern. Table 2.2 and Table 2.3 report the description of the HPC dwarfs.

With the increased relevance of parallel computation also in the general purpose market, the dwarfs list was analyzed and extended, leading to the identification of 6 more dwarfs [18]. As it can be seen in Table 2.4, the description of new dwarfs is more concise with respect to the HPC dwarfs. This is due to the fact that it is not clear which is the best approach to write parallel program involving these dwarfs, but their goal is

Table 2.2: HPC dwarfs and their descriptions (from [18]) – part I

Dwarf	Description
Dense linear algebra	Data are dense matrices or vectors. Generally, such applications use unit-stride memory accesses to read data from rows, and strided accesses to read data from columns
Sparse linear algebra	Data sets include many zero values. Data is usually stored in compressed matrices to reduce the storage and bandwidth requirements to access all of the nonzero values. Because of the compressed formats, data is generally accessed with indexed loads and stores
Spectral methods	Data are in the frequency domain, as opposed to the time or spatial domains. Typically, spectral methods use multiple butterfly stages which combine multiply-and-add operations and a specific pattern of data permutation, with all-to-all communication for some stages and strictly local for others
N-body methods	Depends on interactions between many discrete points. Variations include particle-particle methods, where every point depends on all others, leading to an $\mathcal{O}(n^2)$ calculation, and hierarchical particle methods, which combine forces or potentials from multiple points to reduce the computational complexity to $\mathcal{O}(n \log n)$ or $\mathcal{O}(n)$

Table 2.3: HPC dwarfs and their descriptions (from [18]) – part II

Dwarf	Description
Structured grids	Represented by a regular grid; points on grid are conceptually updated together. It has high spatial locality. Updates may be in place or between 2 versions of the grid. The grid may be subdivided into finer grids in areas of interest, and the transition between granularities may happen dynamically
Unstructured grids	An irregular grid where data locations are selected, usually by underlying characteristics of the application. Data point location and connectivity of neighboring points must be explicit. The points on the grid are conceptually updated together. Updates typically involve multiple levels of memory reference indirection, as an update to any point requires first determining a list of neighboring points, and the loading values from those neighboring points
Monte Carlo	Calculations depend on statistical results of repeated random trials. Considered embarrassing parallel

Table 2.4: New dwarfs and their descriptions (from [18])

Dwarf	Description
Combinatorial logic	Functions that are implemented with logical functions and stored state
Graph traversal	Visits many nodes by following successive edges. These applications typically involve many levels of indirection, and a relatively small amount of computation
Dynamic programming	Computes a solution by solving simpler overlapping subproblems. Particularly useful in optimization problems with a large set of feasible solutions
Backtrack and branch-and-bound	Find an optimal solution by recursively dividing the feasible region into sub-domains, and then pruning subproblems that are suboptimal
Construct graphical models	Constructs graphs that represent random variables as nodes and conditional dependencies as edges. Examples include Bayesian networks and Hidden Markov Models
Finite state machine	A system whose behaviour is defined by states, transitions defined by inputs and the current state, and events associated with transitions or states

perfectly matched: they still point out a set of relevant kernels expected to be employed by next generation applications.

Analyzing the dwarfs, while keeping in mind their instances in the parallel computing world, we can detect a set of relevant problems that should be addressed.

First of all, to guarantee performance improvements, it is mandatory to act on the sequential part of the application. Indeed, Amdahl's law still puts a limit on the maximum obtainable speedup. In the case of parallel applications, synchronization between threads is a well known bottleneck.

Another important issue is represented by efficient data access. Indeed, modern parallel architectures access data in a non-uniform way. To reduce fetching delays, the distribution of data above memory modules

must be exploited to produce a parallel schedule aiming at maximizing the number of access to local memory modules.

When looking at the architecture, from the software point of view, it is clear that handling all possible target devices is not a viable option, and we cannot compile and distribute software only for few architectures, because the features of the device that will execute the software are not guaranteed to be known at compile-time. Adapting to a wide range of architectures has been traditionally tackled by means of *Just-In-Time* compilation, however its cost is still relevant, due to the need of executing a compiler at run-time. This implies that less intrusive techniques must be developed to limit as much as possible the overhead of run-time code analysis and specialization.

Lastly, the increased transistor density has modified the behaviour of architectures with respect to faults: they happen also inside processing units, so we must understand how they affect computation under this new faulting hypotheses. Indeed, from the point of view of parallel software, we know that incorrect synchronization is one of the most widespread reasons of incorrect behaviour, so it is of particular interest to look at what happens when a fault occurs on hardware components responsible of synchronizing the execution of parallel applications.

2.5 Concluding Remarks

In this chapter we proposed a brief survey about parallel architectures, parallel programming models, and workloads stressed by current and future applications.

From the architecture point of view, we have shown that there is no single reference design for parallel computers. Indeed, each design is suitable for a particular execution pattern, which can execute very efficiently. On the other hand, trying to execute patterns not suited for a given parallel architecture results on low performance gains, or even no gain at all.

From the programming model point of view, we have observed a similar scenario. The choice of the correct programming model for an application intended to be run on a parallel architecture is mandatory to achieve performance. We have shown that despite the parallelization scheme adopted by each programming model, there are some shared problems. Indeed, optimization of synchronization primitives is a key point to allow a group of threads to collectively reach a given goal. Moreover, efficient data access plays a central role in modern parallel architectures, so the programming model is responsible of linking, in

the most simple way, code execution with data placement, to allow the hardware minimizing expensive data transfers.

Finally, a set of relevant problems, called patterns, have been introduced. They represent the kernels of current – and future – applications, so if we want to exploit parallelism as a mean for achieving performance, we must take them into consideration.

3 Optimizing Reductions in Shared Memory Multiprocessors

Reduction operations play a key role in modern massively data parallel computation. However, current implementations in shared memory programming APIs such as OpenMP are often computation bottlenecks due to the high number of atomic operations involved. We propose a reduction design that exploits the coupling with a barrier synchronization to optimize the execution of the reduction. Experimental results show how the number of atomic operations involved is dramatically reduced, which can lead to significant improvement in scaling properties on large numbers of processing elements. We report a speedup of 1.53x on the *312.swim_m* over the baseline.

3.1 Introduction

When considering data parallelism, reduction operations are a key component of many algorithms. Typical implementations of the reduction construct fall into three categories: either the reduction is performed in a critical section by a single thread; or atomic read-modify-write instructions are used to concurrently aggregate data; or the availability of fast barrier synchronization is exploited to divide the reduction into two smaller operations, each executed by a different thread. The last case is commonly used, e.g., in GPGPU code [87]. On the other hand, standard benchmark kernels such as *streamcluster* from the PARSEC [26] suite employ the first method. Other benchmark suites employ the reduction support provided by the OpenMP [16] Application Programming Interface (API).

In the case of OpenMP, a `reduction` clause is associated with a parallel loop directive and defines a reduction operation using a combination operator specified in the clause. Support for *reduce*-like constructs is limited to associative and commutative binary operators and, in the case of Fortran, intrinsic procedures, which are also associative and commutative functions. Therefore the reduction loops can be parallelized by associating each thread with a subset of the elements to be combined. The partial reduction value computed by each thread can then be com-

bined in pairs recursively until a single reduction value is produced. This process takes a logarithmic number of steps with respect to the initial number of threads.

The parallel loop implements a *fork-join* model, which requires a single implicit synchronization. In the general case, a single barrier synchronization is needed to ensure that all iterations of a parallel loop are completed at the join point before moving to other parts of the program. This implicit synchronization can be removed with a `nowait` clause, while explicit synchronizations can also be used to handle data dependencies.

On the other hand, the reduction step, which always takes place at the end of a parallel loop, requires more synchronization. This synchronization overhead leads the reduction step to cause loss of scalability, to the point where reduction overhead can become a critical issue, as shown by Fürlinger et al. [57] for the *312.swim_m* SPEC OMP2001 benchmark.

The goal of this work is to introduce an optimized barrier synchronization and reduction step, by allowing the intermediate values of the reduction to be carried along by the inter-thread communication required for the barrier synchronization.

The proposed solution is demonstrated by means of both OpenMP and *pthread-based* implementations. The *pthread* implementation is standalone and introduces a combined barrier-reduction function.

In the case of OpenMP, we replace *libgomp* [1] barrier synchronizations involved in a reduction with a tournament barrier [72], which is both more efficient and scalable, and mirrors the tree structure of the parallel reduction. We then use the atomically-accessible flags of the tournament barrier to store partial reduction values, thus removing the need for locks in communicating the partial values.

An experimental campaign conducted on the reduction benchmarks from most representative suites shows speedups up to 1.53x.

The rest of this chapter is organized as follows. Section 3.2 gives a brief overview of barrier synchronization and reduction implementations state of the art. Section 3.3 provides a detailed description of our solution, while Section 3.4 shows its worth through an experimental campaign on both benchmark applications and synthetic micro-benchmarks. Finally, Section 3.5 provides comparison with the state of the art in reduction optimization and Section 3.6 draws some conclusions and highlights future research directions.

[1] *libgomp* is the OpenMP runtime implementation provided by the GNU GCC compiler [1].

3.2 Background

In this Section, we review the background in barrier synchronization algorithms and parallel reduction implementation, with an eye to the implementation of both features in OpenMP.

3.2.1 Barrier Synchronization

Barrier synchronization overheads account for a large fraction of the communication time in parallel/concurrent applications.

Barriers can be used with both message passing and shared memory programming models. In this chapter, we will describe barrier algorithms in terms of the shared memory programming model, since it is the one implemented in OpenMP.

The goal of an optimized barrier algorithm is in both cases to minimize the communication involved during each barrier operation. In the case of message passing, this is represented by the packets sent, while for shared memory the communication is obtained through the execution of atomic instructions, as their execution is guaranteed to be correctly observed by threads other than the one performing them.

The minimization of barrier synchronization overheads has been addressed by a large number of studies [109] proposing new barrier algorithms. In general terms, we can identify three class of barrier algorithms: *centralized*, *dissemination* and *tree* barrier.

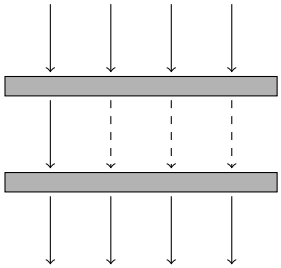
Centralized The barrier state is represented by a shared centralized structure, such as a counter; each thread atomically increments the counter, then it spins over the counter, using atomic operations in the process, until the expected final value is reached.

Dissemination The barrier state is a partitioned into sections, each accessed by a subset of threads using the barrier; splitting the state allows to minimize communication needed to keep a consistent state. In general, more communication operations are needed than in a centralized barrier, but since most communications access different sections of the barrier state, conflicts are reduced, producing an overall reduction in execution time.

Tree The barrier state is partitioned, spread across threads using the barrier and laid out in a tree structure; this results in high memory consumption to maintain a tree data structure, but minimizes both communication and conflicts.

```
1 GOMP_barrier();
2 tid = omp_get_thread_num();
3 if(tid == 0) {
4   red = 0;
5   for(i = 0; i < SIZE; ++i)
6     red += data[i];
7 }
8 GOMP_barrier();
```

(a) Code



(b) Execution

Figure 3.1: Serialized reduction example. After the parallel loop, the master thread aggregates data of all the others. Reduction is constrained between two barrier operations, represented by gray blocks. During reduction computation, worker threads inactivity is represented by means of dashed arcs

The centralized barrier class includes the central counter barrier [55], used in *libgomp*; the butterfly barrier [32] belongs to the dissemination class; the tournament barrier [72] is an example of a tree barrier. A full analysis of the state of the art is beyond the scope of this chapter, but a good survey is provided by Nanjegowda et al. [109].

Distributing the barrier state among threads is a mandatory feature in the message passing programming models – it allows to distribute the communication traffic. However, it is also important in the shared memory programming model, as it allows to reduce the number of invocations of the cache coherency protocols.

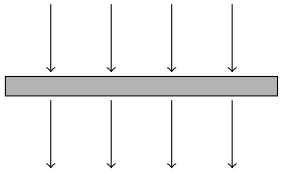
3.2.2 Reduction Implementations

A reduction operation computes a scalar value as a combination of values in a sequence. In a OpenMP parallel region, a reduction is almost always followed by a barrier operation. This allows the reduction value to be correctly seen by all threads after leaving the barrier.

The reduction itself can be executed in several different ways. In the most trivial scheme, the reduction is computed by the master thread between two barrier operations, as depicted in Figure 3.1. The reduction is computed sequentially. The first barrier (Line 1) ensures that the master thread sees a consistent state of the memory – all other threads must have finished the previous phase – before starting aggregating values (Line 5). The second barrier (Line 8) blocks other threads until

```
1 private_red = 0;
2 for(i = lw; i < up; ++i)
3   private_red += data[i];
4 atomic_add(&red,
5            private_red);
6 GOMP_barrier();
```

(a) Code



(b) Execution

Figure 3.2: Parallelized reduction example. Reduction is computed concurrently. Just before the end of the parallel loop, each thread aggregates its local reduction value to the global one by means of atomic operations. Only one barrier operation, represented by a gray block, is executed

the reduction is completed. Such a simple scheme obviously sacrifices all opportunities for parallelization, and involves two barrier synchronizations, but the reduction itself is computed without performing any read-modify-write atomic instruction.

In general, however, the OpenMP compiler parallelizes the reduction. In this scenario, reported in Figure 3.2, the reduction value is a variable shared among all threads. Each thread performs a partial reduction over private data (Line 3), and then safely aggregate the partial reduction value to the global one (Line 5). In addition to parallelization, this scheme allows the elimination of the first barrier. On the other hand, the global aggregation can be performed inside a critical section, or be executed through an atomic read-write-modify instruction – both of which are expensive.

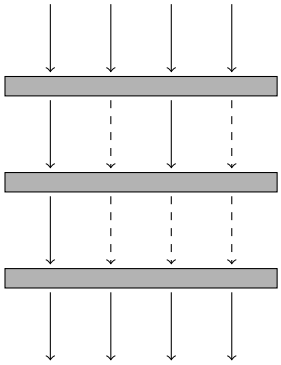
If the hardware architecture supports fast barrier synchronization, it is also possible to perform reductions in a logarithmic number of rounds, using a divide et impera approach, as depicted in Figure 3.3. Rounds (Line 4) are executed in a lock-step fashion, exploiting barriers (Line 7) for coordination. Inside each of them, only a subset of threads are active, the ones that are computing partial reductions (Line 6). At the last round, only one thread is active. It is in charge of computing the final reduction value (Line 10). Since this implementation requires $\log_2(n)$ barrier synchronizations, where n is the size of the sequence, it is only acceptable when there is hardware support for fast barriers.

3.2.3 Atomic Operations

To allow threads to coordinate their execution, modern microprocessors support atomic memory access operations. In some cases, the atomicity

```
1  GOMP_barrier();
2  tid = omp_get_thread_num();
3  sred[tid] = data[tid];
4  for(i = 1; i < SIZE; i *= 2) {
5    if(tid % (2*i) == 0)
6      sred[tid] += data[tid + i];
7    GOMP_barrier();
8  }
9  if(tid == 0)
10   red = sred[0];
11 GOMP_barrier();
```

(a) Code



(b) Execution

Figure 3.3: Hand-written reduction. Under the hypothesis of fast barrier operations, it is possible computing the reduction using a reduction tree. At each round, only some threads are active, the ones that are computing the reduction, represented by solid arcs. Rounds are executed in a lock-step mode. In the last round, only one thread is active, the one computing the whole reduction value

is guaranteed by hardware properties for memory read and write operations. For example, on the Intel x86 P6 family processors every load and store aligned to 8/16/32/64 bits fitting into a cache line is atomic [80].

However, in most cases the atomic operations are more complex than simple reads or writes. The two most popular classes of atomic operations are the *read-modify-write* and the *compare-and-swap*.

Atomic read-modify-write instructions atomically read a value from memory, perform an arithmetic or logic operation, and write the result in the same memory address from which the operand was read. On modern microprocessors, the atomicity is implemented on top of the cache coherency mechanism [70].

Compare-and-swap instructions allow to atomically read a value from the memory, and optionally replace it with the content of an operand. Compare-and-swap operations are more powerful than any atomic read-modify-write instruction [73], but costs are comparable – in both cases, the time spent achieving atomicity is the dominant cost factor.

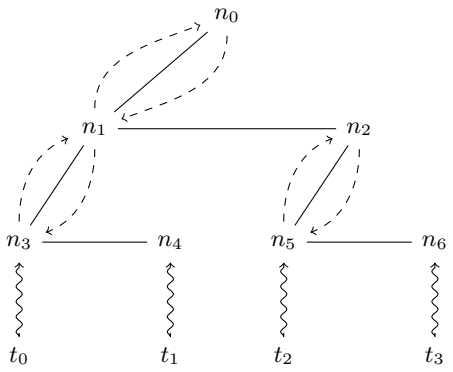


Figure 3.4: Execution of the tournament barrier algorithm. Each thread enters into the barrier via a statically assigned leaf. The dashed path is followed by threads entering in an active node. They climb the tree until a passive node or the root node is reached

3.3 Combining Barrier and Reduction

To mitigate reduction overhead, we can combine the execution of each reduction and its associated barrier. This allows to pay synchronization cost once, while performing two operations – reduction and barrier.

To improve performance, we also aim at reducing the usage of atomic read-modify-write instructions as much as possible. Thus, we choose the *tournament barrier* [72] as a starting point for our reduction design, since it achieves synchronization without performing any atomic read-modify-write instruction [109].

3.3.1 Tournament Barrier

The tournament barrier employs a binary tree data structure, where each of the threads that need to be synchronized is statically associated to an arbitrarily chosen leaf. Thus, for synchronizing n threads, the algorithm uses a tree with $2n - 1$ nodes. The algorithm operates in $\log_2 n$ rounds.

Example 3.1. Consider the four threads and the associated barrier tree shown in Figure 3.4. The barrier tree is a complete binary tree, with four leaves, n_3 to n_6. Odd numbered nodes are *active*, while even numbered ones are *passive*, except for the root node n_0. At the beginning each thread is assigned to a leaf node. Threads t_0 and t_2 enter into active nodes and start spinning until they are signalled by their siblings. Threads t_1 and t_3 enter passive nodes, signal their siblings, and start

spinning until they are notified during the exit phase. Once t_0 and t_2 have been notified by t_1 and t_3, they move to n_1 and n_2 respectively, starting a new synchronization round. In this round t_0 moves to an active node, while t_2 to a passive node. Thus t_0 progresses to the root node n_0, while t_2 waits spinning. Once t_0 reaches the root node, it starts the barrier exit phase. First t_0 returns to n_1 and signals to t_2 to leave the barrier, then it moves to n_3, signals t_1 that synchronization has been performed and leaves the barrier; t_2, in turn, notifies t_3. Once notified, t_1 and t_3 leave the barrier.

The standard tournament barrier avoids atomic read-modify-write instructions by exploiting point-to-point synchronization – each node contains a flag variable, which is written only by its sibling. Thus, each flag variable is only written by a single thread and hence no conflicts can occur.

However, such feature comes at a cost – the tournament barrier consumes more memory than other barrier algorithms. Moreover, the size and alignment of the flag must be carefully chosen to avoid false-sharing – indeed, if two flag variables share the same cache line, every update to one of the two triggers the execution of the cache-coherency algorithm, thus degrading performance. The flag must thus have a size equal to the cache line, even though it only carries one bit of information – all other bits are just padding. E.g., on a machine with a 64-byte (512-bits) wide cache line, each flag includes 511 bits of unused padding.

3.3.2 Basic Reduction Design

The key idea of our design is to exploit the free space available in the tournament barrier flag variable to propagate the partial results of the reduction operation, computing them within the nodes.

To this end, flag variables are stored into the widest type that allows atomic read/write access without locking – we will call this type the *container type* in the rest of the chapter. They are also aligned to the cache line size, to avoid false-sharing.

The container type is split into two sections, shown in Figure 3.5(a): *flag bit* stores the state of the barrier operation (1 bit); *payload* stores the state of the reduction operation ($n - 1$ bits, where n is the size of the container type).

In the case of a 64-bit machine with a 64-byte wide cache line, the container type is a 64-bit integer, aligned to 64-bytes.

As depicted in Figure 3.5(a), the first bit is the flag bit, while the remaining bits of the container type represent the payload.

(a) Base (b) Extended

Figure 3.5: Layout of the container type. In the base version only one bit is needed to encode the barrier state, all others can be used to pack partial reduction values. The extended layout uses one more bit to find whether the reduction partial value is packed into the container payload or stored in the auxiliary variable

All bits needed to align the container to the cache line are wasted, since we cannot access them atomically without using locks and thus adding an overhead that would prevent the algorithm from achieving a speedup with respect to existing designs.

A thread entering a passive node stores into its active sibling both the flag bit and the payload containing its own partial reduction result. Then, it waits to be notified by its active sibling by spinning on its own flag variable. At each spin, the value of the flag bit is extracted from the container and checked.

A thread entering an active node first spins over its flag variable, waiting for the thread associated with the passive sibling node to reach the barrier. At each spin, the container flag variable is read and the flag bit is extracted and checked. If the flag bit is set, the payload is also extracted and aggregated with the private partial result of the thread. The thread then enters the parent node, starting a new round of the algorithm.

When a thread returns to an active node after visiting its parent, the same operations are performed as in the exit phase of the basic tournament barrier algorithm. The thread notifies its passive sibling that synchronization has been achieved by setting the flag bit into its flag variable.

In our design, reaching the root node has a double meaning: not only all threads have reached the barrier, but the reduction is also computed, and its value is stored in the current thread private memory space. To make this value readable to all threads, it is necessary to store it in

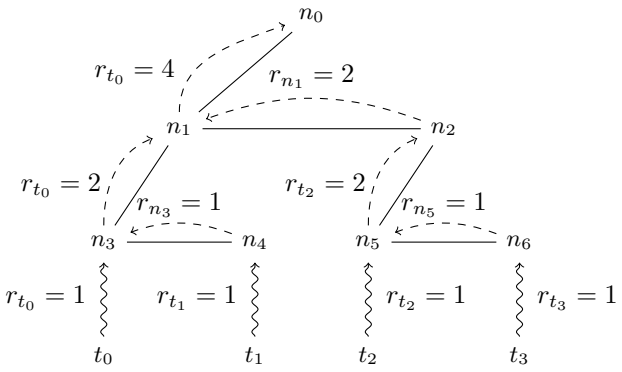


Figure 3.6: An example of reduction. There are four threads, each proposing 1 as the value to be aggregated. The reduction operator is sum. The r_{t_i} variable refers to the partial reduction seen by thread t_i, while r_{n_i} is the value of the partial reduction inside node n_i. The reduction is computed along the dashed path. Partial reductions are computed while moving from a node to its parent. Passive nodes sends their partial reduction values to associated active nodes

a global-accessible variable and then force a memory fence operation. At this point the reduction is completed and the tournament barrier algorithm can proceed, notifying threads that synchronization has been achieved.

Example 3.2. We want to compute the sum of a sequence of unsigned integer values. Assume that the sequence to be reduced has been split into four subsequences, and partial aggregate values have been computed by each of the four threads, as shown in Figure 3.6. Each thread t_i ($i \in [0:3]$) enters the barrier carrying a partial aggregate value p_i. The algorithm performs the same steps as in the standard tournament barrier implementation show in Figure 3.4. In addition, at each step threads in passive nodes pack their partial aggregate value together with the flag value into their sibling node. Therefore in the first step t_1 and t_3 store their partial values p_1 and p_3 into nodes n_3 and n_5. Then, t_0 and t_2 before moving to the second step extract from their respective containers the payload and compute new partial values by aggregating respectively $p_0 + p_1$ and $p_2 + p_3$. In the second step t_2 packs its computed partial value into n_1, where it is extracted by t_0 and combined to obtain the global reduction value $p_0 + p_1 + p_2 + p_3$. This value is published by t_0 when it reaches the root node n_0. The exit phase is unmodified with respect to Example 3.1.

3.3.3 Fast Path Optimization

The basic reduction design represents a *fast execution path*, which is only semantically correct under the condition that the reduction data-type fits the size of the container payload. To handle the remaining cases, a fall-back *slow path* will be introduced in Section 3.3.4.

The efficiency of the fast path strictly depends on the ability of the base tournament barrier algorithm to parallelize the reduction operation as well as to minimize the number of atomic operations. The reduction parallelism is achieved by exploiting the hierarchical structure of the barrier tree, while independence derives from limiting the entities performing the partial reductions to two, namely reader and writer. Thus an *f-way tournament barrier* [65] would not be as effective as a base algorithm for our purpose.

The fast path requires only one memory fence. The thread that reaches the root node performs this memory fence to make the final result of the reduction visible to all threads.

While the ability to take the fast path is dependent on the reduction data type, it is independent from the operator used to aggregate values. As long as partial reduction values fit into the container payload, atomic read-modify-write operations and memory fences can be avoided.

3.3.4 Slow Path Management

The *slow path* is designed as an extension of the basic reduction algorithm to handle the case when the reduction data-type does not fit the container payload.

To this end, the container layout has been further modified, as shown in Figure 3.5(b), to reserve space for a 1 bit field – the *path* field. Consequently, the payload field is shrunk by 1 bit. An auxiliary variable is added to the node state to hold the partial reduction value.

Figure 3.7 shows the pseudo-code of the path management algorithm. When the thread in the passive node needs to propagate the reduction value to the thread associated with the active sibling, the management algorithm is invoked. If the partial reduction does not fit into the payload (Line 2), it is stored into the auxiliary variable (Line 5) associated with the active sibling of the current node. Then a memory fence is issued (Line 6). Finally the $flag$ and *path* fields of the container are set.

Correspondingly, active nodes detect where to read the reduction partial value by reading the path bit of their container. If the path bit is set, the slow path has been taken, and the reduction partial value can be found in the auxiliary variable associated with the active node. Oth-

Algorithm: PathManagement
Input: a partial reduction value *data*
a passive tournament barrier node *node*
Result: reduction information is communicated to the active sibling of *node*

```
1 sibling ← GetSibling (node)
2 if Fits (data, payload_size) then
3   sibling.containter ← Pack (data, FAST, flag)
4 else
5   sibling.auxiliary ← data
6   MFence ()
7   siblig.container ← Pack (0, SLOW, flag)
```

Figure 3.7: Path management algorithm. When the partially reduced value fits the payload, the fast path is taken; otherwise a slow path involving a memory fence is triggered

erwise, the fast path has been executed – the reduction partial value is packed into the payload (Line 3).

Note that the memory fence is necessary to guarantee that the partial reduction value is stored into the auxiliary variable before the flag and path bits are set, but induces an increased latency. Such fence instructions are not issued on reductions performed using the fast path, since in this case the partial reduction values and the flags are written atomically.

Since modern processors are usually 64-bit based, the payload is large enough to hold partial reduction values of most native scalar data-types. Therefore the slow path is rarely taken. In the next Section, we show how to deal with larger data types and still benefit from the fast path.

3.3.5 Compact Data Representation

Taking the fall-back slow path is not always necessary when the data size is too wide by just 2 bits. As an example, consider a reduction over 32-bit unsigned integers on a 32-bit machine. Since we use 1 bit to represent the flag and 1 for the path field, the payload is not wide enough to store a 32-bit unsigned integer. Thus, in the many cases where the values involved in the reduction never exceed $2^{30} - 1$, we could still use the fast path – the same might not be true in the case of signed integers, though.

The packing function used to store the partial reduction values into the payload is therefore parametrized with respect to the reduction data type

and values. When working with the widest unsigned integer type that allows atomic read/write access, the packing function checks whether the value can actually fit into the payload (i.e., the two most significant bits are not set).

In these cases, the algorithm is not forced to take the slow path over all nodes – path selection strictly depends on the actual value of the reduction in each active thread. If a partial reduction value follows a slow path, this does not force a slow path for the other threads. In many cases, such as when a reduction is used to sum partial counters, it is more likely to overflow payload bounds only in the last rounds of the algorithm, which also involve only few threads, thus using a fast path in most nodes of the barrier tree.

To exploit this path optimization in the very common case where reductions are performed over word-size floating point values, we need to recover two bits from the floating point representation, without losing precision. IEEE double precision floating point numbers [79] fp are represented over 64 bits, $\langle fp_{63} \dots fp_0 \rangle$, with the following interpretation: $sign = fp_{63}$ holds the sign, $exp = \langle fp_{62} \dots fp_{52} \rangle$ represent the biased exponent, and all other bits hold the $mantissa$ (except the first digit, which is implicitly set at 1). Thus fp represent the floating point number $(-1)^{sign} \times 2^{exp-1023} \times (1.0 + mantissa)$.

To preserve precision, the algorithm cannot simply discard the least significant bits of the mantissa. We therefore operate in the same way as for the integers, assuming implicit values for two bits. These bits, and the relative assumed values, must be chosen to maximize the execution frequency of the fast path.

The distribution of mantissa bits is hard to predict, and making the sign implicit would limit the fast path to just positive or negative values. Thus, we have to choose two bits from the exponent. Since the exponent is biased, the first two bits of the exponent partition the space of floating point numbers in four equally sized subspaces. The 10_2 subspace contains exponents ranging from 1 to 512, making it a good candidate for the assumed value. The 11_2 subspace represents very large numbers (in modulo), that are expected to appear late if at all in the reduction, while 00_2 represents very small values, which would often be overshadowed by larger values early in the partial computations. Finally, the 01_2 subspace contains exponents between -511 and 0, which makes it an excellent candidate, since it represents most of the range $(2, -2)$ (excluding the values with a modulo close to zero), which is suitable for many computations.

In the end, the choice between 01_2 and 10_2 mostly depends on the application domain. For the experiments reported in this chapter, we

use the 01_2 setup.

3.3.6 Nowait Reductions

Sometimes, it is necessary to aggregate different variables at a synchronization point, and there are no data dependencies among the different reduction operations. In the case of multiple consecutive reductions, we could still use a combined reduction/barrier operation for each reduction operation. However, this scheme enforces some useless synchronizations, as once a thread has reached a passive node and has sent its reduction partial value to its active sibling, it is no longer necessary to wait at the barrier, as synchronization is not actually needed except in the last reduction. We call this kind of reduction a *nowait reduction*.

Nowait reductions are easily expressed within OpenMP programs. Work-sharing constructs can be tagged with the `nowait` clause to avoid a barrier operation before leaving the construct. If a `reduction` clause is also present, our combined barrier algorithm can be executed in nowait mode to compute the reduction value, issuing fewer atomic instructions than the standard implementation.

The base algorithm has been modified to support nowait reductions. A thread t_i reaching a passive node sends the reduction partial value to its sibling t_j, and starts waiting for a synchronization achieved signal. Once notified, thread t_j performs the local aggregation pass, and then releases thread t_i *before* moving to the parent node. This allows t_i to leave the barrier earlier with respect to the base algorithm, and the exit phase is not performed at all.

This scheme keeps only those threads that are actually working to compute partial values of the reduction in the barrier, while all other can proceed to the next program statement. When the following statement is also a combined barrier/reduction operation, reductions are pipelined.

When operating in nowait mode, the thread reaching the tree root does not issue any memory fence, since synchronization is not needed, and so publishing the reduction global value is not mandatory. Consequently, if global synchronization is needed, the last barrier operation cannot be performed in nowait mode.

3.4 Experimental Evaluation

In this Section we provide an experimental evaluation of the proposed technique. While there is no standard suite dedicated to reduction benchmarking, the three most popular suites all include one benchmark specifically chosen to measure the effectiveness of this operation.

Table 3.1: Reduction benchmark characterization

Benchmark	Reductions	Operator	Data type
cg (class C)	3900	+	floating point
312.swim_m	2400	+	floating point
fast	50000	&	unsigned integer
slow	50000	&	unsigned integer
mixed	50000	+	unsigned integer
multi	50000	&	unsigned integer

To supplement these benchmarks, we also employ micro-benchmarks to measure specific properties.

3.4.1 Benchmarks

We select a set of benchmarks from the most popular suites targeting shared memory parallel applications: SPEC OMP2001 [20] and NAS [83]. Fürlinger et al. [57] show the bottlenecks for the SPEC OMP2001 benchmarks. According to their analysis, *312.swim_m* and *310.wupwise_m* are the only benchmark in the suite where reductions have a significant impact, though *310.wupwise_m* uses complex data types, and thus is not optimizable in our framework, since we need to carry the payload in an atomically accessible data type, not just a data type fitting the cache line. NAS also provides a single interesting benchmark, *cg*.

In addition to evaluation on benchmark applications, synthetic micro-benchmarks are useful to analyze the performance properties of the proposed reduction design. The only well known micro-benchmark suite for OpenMP constructs is EPCC [36]. The EPCC *syncbench* benchmark is designed to stress reduction computations. Its kernel is an `omp parallel` region. However, the GCC OpenMP implementation introduces implicit barriers at both region start and end. Since the region body in the benchmark does not perform any relevant computation, GCC-induced synchronizations dominate the benchmark runtime, making it all but impossible to use it for its designated purpose. Moreover, *syncbench* does not help in understanding the behavior of the reduction design. Therefore, we employ four synthetic micro-benchmarks.

Table 3.1 shows the resulting benchmark set, characterized by the dynamic count of reduction operations, as well as by the type of reductions and the data types involved. The set covers all interesting data types:

integers and floating point numbers, in the latter case including both single and double precision.

3.4.2 GCC Optimization

All benchmarks are parallelized exploiting OpenMP directives. Thus, we have introduced compiler support for our combined barrier and reduction implementation in GCC. When a reduction clause that can be optimized is found, a GCC optimization pass identifies the barrier operations executed after the reduction, and replaces both with the invocation of our combined reduction and barrier. To this end, we have also augmented the GCC OpenMP runtime, *libgomp*, with our barrier implementation. To measure the efficiency of the combined barrier and reduction (and not the efficiency of the barrier alone), we still rely on the default barrier implementation (a central counter barrier) in all cases except those where the combined barrier and reduction is used.

3.4.3 Experimental Setup

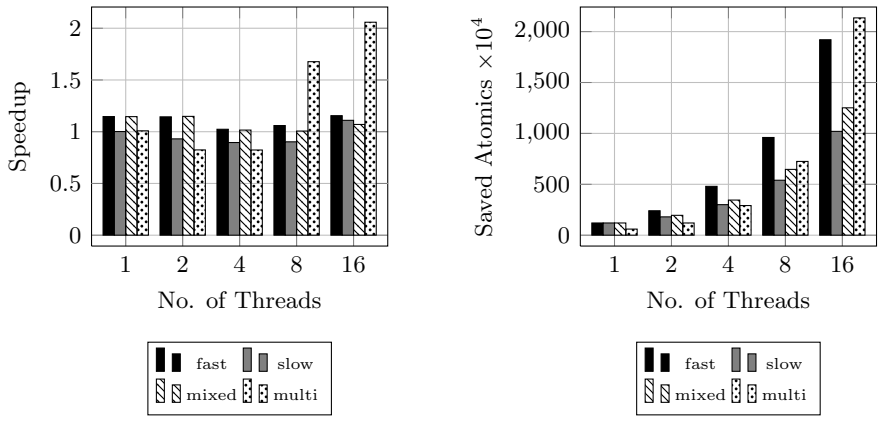
The experimental campaign has been conducted on a AMD NUMA machine with four nodes, each a quad core Opteron 8378 processor. Each core has a 64KBytes L1 data cache, a 64KBytes L1 instruction cache, and a unified L2 512KBytes cache. All cores within a node share an unified 6MBytes L3 cache. Cache line size is 64Bytes Inter-node communication is supported by a fully-connected network.

All benchmarks are compiled with the GCC 4.6 compiler in two flavors: *base* and *peak*. Base compilation is the reference execution obtained using an unmodified GCC compiler and runtime, while peak compilation applies optimization to use our combined reduction-barrier.

For each flavor, we register both the execution times and the number of atomic operations performed. All benchmarks are run with a number of threads varying from 1 to 16 – the maximum available hardware parallelism.

3.4.4 Micro-benchmarks

The *fast* micro-benchmark stresses the execution of the fast path. Conversely the *slow* micro-benchmark always triggers the slow path. The *mixed* micro-benchmark evaluates the case where execution starts from the fast path and then triggers the slow path. Finally, the *multi* micro-benchmark targets the nowait reduction behavior in the case of multiple reductions in the same loop. Figure 3.8 summarizes micro-benchmarks results.



(a) Execution time

(b) Atomic operations

Figure 3.8: Reduction micro-benchmarks results

In all cases, we compare our design with the *libgomp* baseline. The results show that for the fast path and the multiple nowait reductions the number of atomic operations is very low and scales well over a larger amount of threads. However, the *multi* micro-benchmark shows that greater benefits are achieved when nowait reductions are involved since in this case our design significantly reduces the amount of synchronization, thus obtaining a major performance improvement over the *libgomp* baseline.

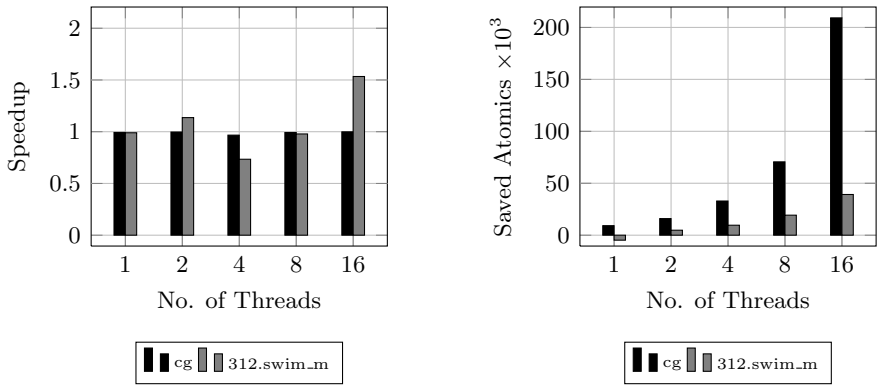
On the other hand, the *slow* and *mixed* micro-benchmarks show that our design does not significantly degrade performance even when the slow path is triggered.

3.4.5 cg

The *cg* benchmark computes the eigenvalues of a sparse matrix using the conjugate gradient method [83], relevant to the field of computational fluid dynamics. The structure of the code includes a top-level loop that contains an OpenMP parallel region. The parallel region computes the aggregate value used in subsequent loop iterations by means of a reduction at the end of the region. It is important to note that an `omp master` construct is used to compute an intermediate aggregate value, and an explicit barrier is used to block all other threads.

We have executed the benchmark using the C data set.

Even if our reduction implementation is effective in reducing the number of issued atomic operations, as shown in Figure 3.9(b), the run-time



(a) Execution time (b) Atomic operations

Figure 3.9: Reduction benchmarks results

is dominated by the `omp master` sections, thus run-times do not scale well.

3.4.6 312.swim_m

This benchmark numerically solves a shallow water modelling problem relevant to weather prediction [20]. It repeatedly executes a computationally intensive loop body containing a parallel OpenMP region. At the end of the parallel region three reductions are computed.

Our algorithm generates two nowait reductions followed by a combined reduction-barrier.

As shown in Figure 3.9(a) and 3.9(b) the number of atomic operations performed by the peak version is always lower than the baseline, while run-times are lower or equal to the baseline when working with more than 4 threads.

When working with only few threads, atomic operations are often uncontested. Thus, the base implementation can outperform the peak implementation in these cases. On the other hand, when the number of threads grows the performance of the peak optimization stabilizes and are always better than the baseline.

3.5 Related Work

In the context of distributed computing, where communication is more expensive than in shared memory architectures, the MPI standard [103]

includes the notion of *collective operations* to perform multiple operations in one step, and reduce the number of exchanged messages.

Shirako and Sarkar [132] introduce the concept of *phaser accumulator* to combine reduction and barrier operations in presence of dynamic parallelism. They rely on atomic read-modify-write instructions to send reduction partial values safely to the phaser. They also propose a tree-like structure for phasers, with the goal of improving phaser scalability [131].

The key difference between our work and the one by Shirako and Sarkar [131] is that we target a static workload and focus on enabling the fast path as much as possible through compact data representation. Shirako and Sarkar [131], on the other hand, focus on the ability to scale dynamically, and therefore the synchronization tree must be specified. Pipeline reduction is another optimization that is only available in our design.

Shirako and Sarkar [131] compare their work with the OpenMP runtime using the EPCC Syncbench [36]. This makes comparison with our work difficult both because of the characteristics of the EPCC Syncbench described in Section 3.4.1 and the different goals, as our work is geared towards a language agnostic reduction design.

Intel TBB [5] employs a similar structure to Shirako and Sarkar [131], with a costly dynamic creation of the reduction tree.

Chun and Xuejwen [41] address the optimization of barriers and reductions with a different approach – rather than handling the fast paths at runtime, they rely on new primitives for expressing constrained forms of barrier and reduction constructs.

3.6 Concluding Remarks

In this chapter, we have proposed a reduction design to take advantage of coupling with a barrier synchronization. Our design exploits the unused space in the flag variables of a tournament barrier to carry a partial reduction value, thus reducing the amount of atomic operations.

Our experimental campaign shows a significant reduction in the number of atomic operations employed to perform the reductions, as well as a speedup of 1.53x on the *312.swim_m* benchmark.

Future directions for this research include affinity-guided association of threads to barrier tree leaves, as well as an adaptive data-compaction method to increase the frequency of the fast path further.

4 Data-aware Iterations Scheduling in OpenMP

In modern NUMA architectures, preserving data access locality is a key issue to guarantee performance. We define, for the OpenMP [16] programming model, a type of architecture-agnostic programmer hint to describe the behaviour of parallel loops. These hints are only related to features of the program, in particular to the data accessed by each loop iteration. The runtime will then combine this information with architectural information gathered during its initialization, to guide task scheduling, in case of dynamic loop iteration scheduling. We prove the effectiveness of the proposed technique on the NAS parallel benchmark suite [83], achieving an average speedup of 1.21x.

4.1 Introduction

To achieve performance in NUMA architectures, it is essential to provide *data access locality*, that is, data located in a given node are accessed as much as possible from the cores of the same node, and as little as possible from the other ones [100, 140]. Recent works targeting OpenMP on Linux focus on exploiting specialized page allocation policies [27] such as explicit data distribution, which allows the programmer to select a precise distribution to be implemented at initialization time. The *next-touch* policy, introduced in [34, 61], allows dynamic data relocation by exploiting memory protection mechanisms.

However, such works incur in one or more of the following drawbacks: they rely on programmer knowledge of the underlying architecture, thus negating a major benefit of OpenMP, architecture independence [111]; they lack dynamism, since they provide only a single data distribution strategy which might not cover all the access patterns the program employs during different phases of its execution; or, they do not deal with workload balancing, which in turn adversely affects irregular parallel applications.

In this work, we take into account these issues, providing a solution to maintain thread-data affinity across the lifetime of the application, which relies on programmer hints describing only the application behavior, and

exploiting them through a specialized runtime, balancing the workload by means of work-stealing.

The rest of this chapter is organized as follows. Section 4.2 introduces the syntax and semantics of the proposed hints, while Section 4.3 provides details on our runtime design and implementation, and Section 4.4 provides an experimental evaluation. Finally, Section 4.5 provides a brief survey of related works, and Section 4.6 draws some conclusions and highlights future research directions.

4.2 The Data Access Pattern Approach

The OpenMP standard provides support for parallel loops through the `omp for` and `omp do` directives [1]. The parallel loop syntax is restricted to force the loop bounds to be loop invariants, since the runtime must always be able to evaluate the iteration space. Once the iteration space has been computed, iterations are first grouped into *chunks* [2] and then mapped to the active threads of the parallel team according to the scheduling policy implemented by the runtime. Programmers can influence the behaviour of the runtime system only by forcing a iteration scheduling policy and specifying a minimum chunk size.

Even though OpenMP allows the programmer to choose among different scheduling strategies, to address the problem of mapping iterations over the threads in a team, there is no support for expressing thread-data affinity [33, 126].

The key idea of our approach is to allow the runtime to identify the portion of data which will be accessed by the iterations of a parallel loop. These iterations will then be scheduled to threads according to a novel dynamic scheduling policy, which will try to preserve locality as much as possible.

To this end, we extend the existing OpenMP parallel loop directive through a new clause representing the *data access pattern*, that is the way loop iterations access the data. The runtime will then use the thread-data affinity information derived from the data access pattern to improve the existing dynamic iteration scheduling policy, by scheduling threads on the cores nearest to the memory where the related data are stored. While automated approaches to page placement do not require changes to the API, identifying and exploiting thread-data affinity at

[1] `omp for` and `omp do` model the same type of parallel loop, in C and Fortran respectively. For brevity in the rest of the paper we will refer to `omp for` but the same considerations apply to `omp do` as well.

[2] We use the term *chunk* to refer to a set of iterations as specified in OpenMP [16].

$$
\begin{array}{lcl}
\mathit{Axiom} & \rightarrow & \texttt{pattern(}\ \mathit{Clause}\ \texttt{)} \\
\mathit{Clause} & \rightarrow & \mathit{DataStructure}\ \texttt{[}\ \mathit{PESeq}\ \texttt{]} \\
\mathit{PESeq} & \rightarrow & \mathit{PESeq}\ \texttt{,}\ \mathit{PatternExpr} \\
 & | & \mathit{PatternExpr} \\
\mathit{PatternExpr} & \rightarrow & \mathit{RangeExpr} \mid \mathit{SliceExpr} \\
\mathit{RangeExpr} & \rightarrow & \mathit{Expr}\ \texttt{:}\ \mathit{Expr} \\
 & | & \mathit{Expr} \mid \texttt{*} \\
\mathit{SliceExpr} & \rightarrow & \texttt{\^{}}\ \mathit{Expr}
\end{array}
$$

Figure 4.1: Pattern clause syntax. *Expr* is any expression of runtime constants, while *DataStructure* can be any array or pointer variable name

compile time might not be feasible, and is in general a very complex task [33]. By contrast, a skilled programmer is able to identify more effectively the patterns used by threads when accessing data, and thus provide precise hints to the runtime. This is, anyway, mandatory if a fine-tuning of the application performances is desired [27, 110, 137].

A key difference with respect to previous works [27], including *PGAS* languages [14, 125], is that to minimize the programming efforts when writing parallel programs, our approach does not rely on explicit data distribution and exploitation of the processor space.

4.2.1 Data Access Pattern Definition

A data access pattern binds iterations in a parallel loop with the portion of memory accessed at runtime. We formally define the data access pattern and the OpenMP syntactic extension needed to support it as follows.

Definition 4.1. A *data access pattern* is an equivalence relation over the elements of a k-dimensional array data structure. An equivalence class under the data access pattern relation is called *tile*. Data access pattern relations are described by means of *pattern clauses*, defined by the grammar in Figure 4.1 and its associated semantics.

In our OpenMP extension, a *pattern clause* (or, for brevity, a pattern) is associated to a loop directive. The first argument of a pattern clause is a reference to the shared data structure that is concurrently accessed by iterations in the loop. The rest of the pattern clause consists of a sequence of *pattern expressions*, one for each dimension. A pattern expression can be either a *range expression* or a *slice expression*. A range expression is used to identify a range of indices in a given dimension of

```
1 #pragma omp for collapse(2) \
2  pattern(A[^RSLICE,^CSLICE])
3 for(i = 0; i < ROWS; i += RSLICE)
4   for(j = 0; j < COLS; j += CSLICE)
5     for(k = 0; k < RSLICE; ++k)
6       for(h = 0; h < CSLICE; ++h)
7         A[i+k][j+h] = ...;
```

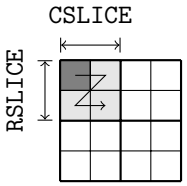


Figure 4.2: Pattern example. Matrix `A` is accessed in a block-wise fashion by the collapsed parallel loop

the data structure, that are associated to all tiles. A slice expression identifies the size of each tile in a given dimension.

A range expression has the form `[n:m]`. Both `n` and `m` must be loop invariant. Their value is thus known at runtime before the loop execution starts. The lower bound of a range expression may be omitted when it matches exactly the lower bound of the associated dimension. Hence, a pattern expression `m` is an alias for `[lb:m]`, where `lb` is the lower bound of the index for the dimension considered. The `*` operator is also a shorthand for `[lb:ub]`, where `lb` and `ub` are the lower bound and the upper bound values of the index for a given dimension. The latter range expression variants allow a more compact definition of the pattern clause in many practical cases, but do not add any expressive power.

A slice expression takes the form `^n`, where `n` is a runtime constant.

Figure 4.2 demonstrates the data access pattern semantics. The two slice expressions define bi-dimensional tiles of size `RSLICE` $\times$ `CSLICE` on matrix `A`, thus representing the block-wise accesses performed by the loop nest.

The mapping between tiles and iterations is defined as follows: if there is no slice expression in the pattern there is a single tile which is accessed by all iterations; otherwise, tiles and iterations are associated by a bijective relation, that depends on both the iteration indices and the sign of the loop increment expressions. In a normalized loop nest, each slice expression is associated to one loop index i_l and the tiles can be ordered with respect to the indices i_d of the dimension d associated to the slice expression divided by the tile size n. Iterations of loop index i_l are mapped to tiles with $i_d/n = i_l$.

Back to the example in Figure 4.2, assuming `RSLICE` = `CSLICE` = 2, and `A` a 4×4 square matrix, the pattern identifies four tiles. The iterations with index `i` = 0 are associated to data items of indices 0 and 1 on the first dimension. The same holds for loop index `j`, which is associated to

the slice expression corresponding to the second dimension of `A`. Thus, iteration `i, j` $= \langle 0, 0 \rangle$ is mapped to the data in `A[0][0]`, `A[0][1]`, `A[1][0]`, and `A[1][1]`.

4.3 Runtime Extensions to Exploit Patterns

To employ the information encoded in the pattern clauses, we propose an extension of the OpenMP runtime. The runtime analyzes each pattern expressions to identify the size of the memory tiles accessed by iterations. The tile information can then be exploited at runtime to group together iterations that will probably touch the same set of virtual memory pages. Since at runtime the base address of the patterned data structure is known, it is always possible to identify the set of memory pages that are expected to be touched by the iterations of the loop. This is true also for dynamically allocated data-structures for which the size can be assumed equal to the tile size times the size of the iteration space.

Since the runtime aims at maximizing the number of local accesses, while avoiding, if possible, to incur in the penalty of long latency due to remote memory accesses, the information obtained analyzing pattern clauses is used to identify groups of iterations (*blocks*) that need to be scheduled together on the same node. Iterations that access the same memory pages (or different pages physically mapped to the same node) are grouped within the same block. The dynamic scheduling policy is thus driven by the collected pattern information.

The implementation used in this work is based on the *libgomp* [1] OpenMP runtime and uses the Linux NUMA API [88] to detect virtual page mappings.

4.3.1 Iteration Space Partitioning

To exploit the hints provided by the pattern information, the runtime has to partition the iteration space so to minimize the number of remote accesses.

Finding an optimal partition is known to be NP-complete. Obviously, such complexity cannot be handled at runtime even with moderate numbers of iterations. Therefore, we propose a straightforward heuristic approach to minimize the time spent by the runtime in analyzing pattern information while still providing a good, even if potentially sub-optimal, partitioning. To further reduce the overhead, we base the partitioning of the iterations of each loop on the information obtained from a single pattern.

The algorithm implemented in the proposed heuristic approach performs a linear scan of the iteration space in search of opportunities for grouping adjacent iterations. Let a and b be two adjacent iterations of the analyzed parallel loop. Both a and b will be mapped to the same block if at least one of the following conditions is satisfied: iteration a accesses to the same set of memory pages touched by iteration b; the set of pages touched by a are physically mapped to a node that is the same for the pages touched by b; pages touched by both a and b are not physically mapped to any node in the system.

Let us now formally introduce the concept of iteration block.

Definition 4.2. Let lb and ub be respectively the lower and upper bound of the iteration space I of the analyzed loop. A block of iterations is defined as a range of indices of the form $[base, last]$, where $base \geq lb$ and $last \leq ub$.

Let B be the set of blocks obtained from the partitioning phase, and let $b \in B$ be a block of iterations. We call $r(b)$ the range of indices described by b.

The runtime limits the maximum number of blocks to reduce the algorithm complexity while maintaining the required flexibility to cope with irregular workloads. The limit has been set, considering the outcome of an experimental campaign, to twice the number of available nodes in the system. To cope with the imposed constraints, different blocks of iterations may be merged.

When no pattern clause is specified for a given parallel loop, the iteration space is evenly partitioned into a number of blocks equal to the number of available nodes. Since the output of the partitioning algorithm is not necessarily the optimal partition, we later introduce a runtime work stealing mechanism to reduce the effects of an unbalanced distribution of the workload.

4.3.2 A Pattern Enabled Dynamic Scheduler

At the end of the partitioning stage, the iteration space of the parallel loop is divided into blocks of iterations. When a loop has associated pattern information, the runtime knows exactly which pages are touched by each iteration block. The runtime assigns a *work queue* to each NUMA node. The work queue is used to store information about iteration blocks. A global work queue is reserved for those blocks that are not related to any of the active NUMA nodes.

The algorithm that maps blocks to work queues uses the iteration-data affinity information coming from the analysis of the pattern. Each

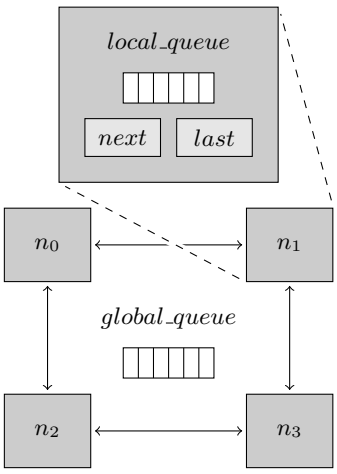


Figure 4.3: Runtime system with four distributed work queues and a global queue

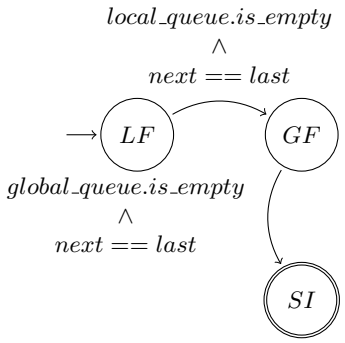


Figure 4.4: Runtime behaviour of a sub-team. *Local Fetch* (*LF*): fetch blocks from the *local_queue* of the local node; *Global Fetch* (*GF*): fetch blocks from the *global_queue*; *Steal Iterations* (*SI*): steal blocks from the *local_queue* of neighbour nodes

thread of the parallel team analyzes the set of blocks in parallel. Let b be a block and let P_b be the set of pages touched by iterations of b. The algorithm counts how many pages in P_b are mapped to each node. The node with the highest number of mapped pages is finally selected as the target node for the block b. If none of the nodes is related to any of the pages in P_b, b is assigned to the global queue.

Figure 4.3 shows the internal state of the runtime system in the case of a ccNUMA architecture with four nodes. The internal state of each node is composed of a working queue called *local_queue* and two integer fields *next* and *last*, used respectively to store the lower bound and the upper bound index of the range of iteration indices associated to the *current block*.

At runtime, parallel teams are split into sub-teams, each associated to a distinct NUMA node. A sub-team associated to a node n is composed only by threads of the team that are running on node n. Threads are mapped to sub-teams at runtime when a new team starts. The runtime behaviour of a sub-team can be formally described by a finite state automaton as shown in Figure 4.4.

Each sub-team starts executing in the initial state LF. Threads of a sub-team whose working state is LF, are only allowed to fetch blocks from the local work queue of the node in which they are running. When

the local block queue is empty and no iterations are available in the current block, the sub-team moves from *LF* to *GF*, where the sub-team fetches blocks from the global queue. In both states, iterations are selected using the Guided Self Scheduling algorithm [121].

The idea is to exploit the locality of accesses preventing when possible threads from accessing remote pages. To this end, at first threads are forced to execute iterations from the local queue to maximize the probability of local accesses. Only when there are no more iterations associated to the local node, threads start fetching iterations from the global queue. Since global queue only stores blocks related to virtual pages that are still not mapped, there is an high probability that threads accessing the global queue will own those pages because of the *first-touch* policy implemented by the OS. The first-touch policy is the default policy for NUMA-aware Linux systems. It consists of placing memory pages on those nodes that first access the data during the program execution.

4.3.3 Work Stealing Strategy

When the global queue is empty and there are no iterations available in the local queue, the sub-team transitions from *GF* to *SI*. While in *SI*, threads start stealing blocks of iterations from the queues associated to other nodes. According to the implemented work stealing policy, threads in *SI* start stealing from the work queues of the nearest neighbour nodes. Since the runtime is aware of the distance between nodes (identified by means of calls to the Linux NUMA API), each sub-team knows which nodes are the best candidates for stealing.

The work stealing procedure iterates over the neighbours set of a node n in search of available blocks of iterations. By default the current neighbour node (n_{neigh}) is initially set equal to the node that hosts the current sub-team (n).

As long as there are iterations to fetch from the work queue of n_{neigh}, threads fetch new iterations from their work queues. Eventually, when the work queue of the current neighbour becomes empty, a new neighbour is selected.

The selection strategy is based on the *NUMA distance* between nodes of the underlying architecture. In the case described in Figure 4.3, $\langle dist(n_0, n_i) | i \in [0:3] \rangle = \langle 0, 1, 1, 2 \rangle$. The distance relation $dist(n, n_i)$ imposes a partial ordering of the nodes $n_i \in K$. We need, for each node, a sequence of nodes to poll for the next neighbour, called a *neighbours vector*. To obtain the vectors, we make this a total ordering by imposing that, when $dist(n, n_i) = dist(n, n_j)$, $n_i \prec n_j$ if $i < j$.

Table 4.1: OpenMP benchmarks characterization

Bench	Parallel loops	Dynamic loops	Patterns
bt.c	28	14	9
cg.c	18	16	16
ft.b	8	6	6
is.c	9	2	2
lu.c	26	10	9
mg.b	14	11	11
sp.c	33	20	20
ua.c	68	56	56

4.4 Experimental Results

In this Section, we provide an experimental validation of our approach. The main findings are that the proposed approach based on pattern clauses is able to consistently reduce the number of remote memory accesses, and that the reduction directly translates into a significant performance improvement.

The experimental campaign has been conducted on a AMD ccNUMA machine with four nodes, each a quad core Opteron 8378 processor. Each core has a two-level private cache hierarchy. L1 cache is composed by a 64KBytes data cache and by a 64KBytes instruction cache. L2 cache is an unified 512KBytes cache. All cores within a node share an unified 6144KBytes L3 cache. Inter-node communication is supported by a ring network topology.

AMD event based counters have been used to measure memory accesses. Separate runs have been used for performance and memory access profiling, to avoid memory access counter sampling overhead in timing measurements.

4.4.1 Benchmark Suite

We employ the NAS Parallel Benchmark suite, OpenMP version 3.3 [83]. We do not report on *DC* and *EP*, since these benchmarks do not have any OpenMP loop constructs (`omp for` and `omp do`). The benchmarks have been modified in order to make use of dynamic scheduling. Table 4.1 shows the number of total loops, dynamically scheduled loops, and loops tagged with the `pattern` clause.

We compare the baseline *libgomp* runtime implementation opportunely

Table 4.2: OpenMP benchmarks runtime behaviour

Bench	Blocks fetched from			Time in opt. loops
	Local [%]	Global [%]	Steal [%]	[%]
bt.c	65.72	0.01	34.27	90.55
cg.c	99.61	0.03	0.36	87.26
ft.b	76.40	0.00	23.60	66.69
is.c	66.67	0.00	33.33	51.30
lu.c	80.21	0.21	19.58	26.49
mg.b	35.16	22.26	42.58	66.82
sp.c	70.03	0.00	29.97	91.92
ua.c	88.36	0.13	11.51	78.28

extended to support a Guided Self Scheduling strategy for dynamically scheduled loop iterations with our optimized runtime. This choice is dictated by the fact that the *libgomp* dynamic scheduler provides only poor performance, thus comparing with it would result in a significant bias due to Guided Self Scheduling.

For all experiments we use 16 threads, each pinned on a different core.

4.4.2 Performance Analysis

Table 4.2 describes the runtime behaviour of the benchmarks, showing the percentage of blocks fetched in each of the states of the automaton in Figure 4.4 along with the percentage of the execution time spent in loops tagged with `pattern` clauses. A high percentage of fetches from local queues denotes a good distribution of the data structures, which is effectively exploited by the iteration scheduling thanks to correct pattern information. On the other hand, blocks fetched through work stealing have higher probability of resulting in remote accesses since they were originally intended to be executed on a different node.

Table 4.3 shows the speedups obtained by our optimized runtime with respect to the baseline. Two scenarios are provided: *Best*, where the proposed work stealing policy based on NUMA distances is used; and *Worst*, where neighbour vectors are reversed. This shows that the order of the neighbours counts: the last column (Δ) shows the maximum performance loss in case of random neighbours selection. However, the results also show that the impact of this policy is not so large as to make the runtime less effective than the baseline. Thus, the *Worst* scenario

Table 4.3: OpenMP benchmarks speedups

	Speedup		
Bench	**Worst**	**Best**	**Δ**
bt.c	1.14	1.27	0.13
cg.c	1.81	1.82	0.01
ft.b	1.12	1.19	0.07
is.c	1.00	1.00	0.00
lu.c	1.02	1.05	0.03
mg.b	1.00	1.00	0.00
sp.c	1.18	1.23	0.05
ua.c	1.07	1.08	0.01

shows the impact of the iteration scheduling optimization, while the *Best* scenario adds the impact of an effective work-stealing policy.

We can see that, for most benchmarks, we obtain a speedup between 1.05x and 1.27x for the *Best* scenario. There are three exceptions: *MG*, *IS* and *CG*.

MG is the only benchmark where the initial distribution of frequently accessed data structures is performed by the master thread alone. Since we rely on the first-touch policy to provide the initial distribution, a large number of remote accesses is generated regardless of the iteration scheduling policy. Note that the pattern definition leads the runtime to place most of the iterations on the node where the master thread resides, thus leading to a reduced amount of blocks fetched from local queues.

IS benchmark implements a bucket sort algorithm. Excluding the time spent in initializing data structures, most of the time is spent on a fast data parallel loop used to sort keys of each bucket. There are several instances of non-linear accesses where array indices are obtained from table lookups. This type of access cannot be optimized, since it is by design hard to predict, to provide the required randomness. While the proposed technique cannot obtain a speedup, it still does not impose an overhead with respect to the baseline.

CG obtains the highest speedup, a remarkable 1.82x. It performs sparse matrix multiplication, which can easily lead to irregular accesses. However, the benchmark provides an initial data distribution that combined with the data access pattern information allows a massive improvement in data access regularity, which immediately translates into a performance improvement.

Table 4.4: OpenMP benchmarks remote accesses (in millions)

Bench	Base	Pattern	Savings [%]
bt.c	70,777.56	54,787.53	22.59
cg.c	40,969.86	3,592.73	91.23
ft.b	4,824.42	4,494.31	0.90
is.c	851.71	844.01	6.84
lu.c	37,674.95	32,731.91	13.12
mg.b	2,504.46	2,486.34	0.72
sp.c	269,485.03	192,971.48	28.39
ua.c	115,912.76	85,196.04	26.50

4.4.3 Remote Memory Access Analysis

Table 4.4 shows the reduction in remote memory accesses obtained by our runtime with respect to the baseline. Memory access reduction is at the base of performance improvement, so these results mirror the performance speedups.

It is especially interesting to consider the reduction in *CG*, where remote memory accesses are strongly minimized thanks to the pattern information.

In *IS*, the data access patterns are mostly unpredictable, as memory accesses are defined through non-affine array functions. This makes it hard to find good pattern information for most of the parallel loops in the code. While the savings in terms of remote accesses are small, they are sufficient to offset the overhead imposed by the pattern evaluation and iteration space partitioning phases.

In *MG*, most frequently accessed data structures are allocated on a single node, which forces all threads on other nodes to perform remote accesses. Thus, no significant reduction is obtained. Moreover, the high amount of global fetches shows that part of data structures were not preallocated at all.

4.5 Related Work

Several different approaches are proposed in literature to mitigate the memory latency penalty due to remote accesses. Some of these approaches rely on the ability of the runtime system [34] or the OS itself [82] to implicitly trigger the migration of worker threads to avoid the cost of remote accesses.

Other approaches, such as PGAS languages [14, 125] rely on the ability of the programmer to manually distribute data structures concurrently accessed by threads at runtime. These languages provide the programmer a mean to force a specific dynamic page placement policy for those shared data structures that will be heavily accessed by loops. On the other hand, our solution does not rely on explicit distribution hints, though it can take advantage of an initial data distribution provided by means of the *first-touch* policy.

Dynamic data distribution based on memory protection mechanisms has been introduced in [34, 61, 93]. Memory pages forming shared data structures can be dynamically tagged, to trigger a page migration to the next node touching them (*next-touch* strategy). Our approach is orthogonal with respect to this strategy, since we reduce the number of remote accesses without triggering redistributions.

In [111] the authors propose a dynamic data redistribution solution similar to [34, 61] but based on information akin to our proposed data access pattern, which is, contrary to our solution, computed at runtime by means of profiling.

4.6 Concluding Remarks

We propose an optimized OpenMP runtime design for NUMA machines to exploit thread-data affinity in parallel programs by means of programmer hints that take into account only the application behavior. Our experimental campaign shows a reduction in the number of remote accesses for most NAS benchmarks.

The approach could be further improved by removing unnecessary pattern evaluations when multiple subsequent loops share the same pattern. Moreover, opportunities for data redistribution could be automatically detected at compile-time by analysing pattern variations between subsequent loops.

Future extensions could include adding thread migration to handle the cases of multiple concurrent applications as well as the case of applications with multiple phases, alternating I/O bound phases with CPU bound ones. We also expect that combining our technique with a next-touch strategy would further reduce the remote accesses, while limiting the number of pages moved.

Furthermore, identifying patterns requires skill and time. It would be worth exploring both static analysis and profiling based techniques to provide recommended patterns to the programmer.

5 Task Assignment in Data Intensive Scalable Computing

MapReduce and other Data-Intensive Scalable Computing paradigms have emerged as the most popular solution for processing massive data sets, a crucial task in surviving the "Data Deluge". Recent works have shown that maintaining data locality is paramount to achieve high performance in such paradigms. To this end, suitable task assignment algorithms are needed. Current solutions use round-robin task assignment policies, which was shown to yield suboptimal results. In this paper, we propose and evaluate new algorithms for task assignment on a model of the Hadoop framework, comparing them with state-of-the-art solutions proposed in theoretical works as well as with the current Hadoop task assignment policies.

5.1 Introduction

The data-intensive computing paradigm has recently received significant attention in both research and industrial ICT communities due to the exponential increase of data available for analytical processing–the so-called "Data Deluge" [77]. The cloud computing scenario represents the most important arena where the potential impact and the effectiveness of data-intensive computing are most visible. The Cloud is an abstraction for the complex infrastructure underlying the Internet and refers to both the applications delivered as services over the network and the hardware and software resources that provide those services. As a key concept, the cloud computing paradigm shifts data storage and computing power away from the user endpoints, across the network, and into large clusters of machines hosted by cloud providers (e.g., Amazon, Google).

The research challenges aimed at exploiting the full potential of data-intensive computing lie in designing clusters and software frameworks to improve performance of massive simultaneous computations, energy efficiency, and reliability of the provided services. In this regard, *MapReduce* is the leading software framework, composed of both a programming model and an associated run-time system, introduced by Google

in 2004 to support distributed computing on large data sets, through splitting the workload over large clusters of commodity PCs [48, 49].

A critical issue to achieve good performance on large scale *MapReduce* systems lies in ensuring that as many data accesses as possible are executed locally. To this end, a data processing job is parallelized in a set of tasks, which are assigned to servers which will execute them. However, purely locality-based scheduling may lead to long latencies, since a specific computation may access data stored on busy servers. Thus, locality-aware, latency minimizing scheduling algorithms have been designed [53] to reduce latency while still exploiting locality.

In this chapter we present an algorithm for task assignment on a cluster of servers that balances latency and resource usage, while also taking into account the workload running on the target cluster. The proposed algorithm is able to achieve an efficient trade-off between latency and resource usage through employing a novel heuristic technique. A simulation-based analysis of the performance of the proposed algorithm against the state-of-the-art solutions is presented, showing that it is able to obtain lower latencies than the standard locality aware round-robin strategy [138], as well as lower resource consumption than the flow-based algorithm reported in [53] together with a better computational complexity. Moreover, we show that our algorithm and the flow-based one are Pareto-optimal with respect to latency and resource consumption, while the round-robin is not.

On the other hand, we does not deal with fault tolerance in MapReduce systems. While this is also a critical issue in achieving performances, it is a different issue from load balancing, which is best covered with specialized approaches that act during the task execution rather than at task assignment. We also do not deal with job scheduling, and therefore with fair-share scheduling among users, as this goal is better achieved at the level of job scheduling.

The remainder of the chapter is organized as follows. Section 5.2 reports a brief summary of background information about MapReduce systems. Section 5.3 defines the abstract model on which the proposed task assignment algorithm is designed. An operative description of the algorithm as well as the description of its properties are also reported. Section 5.4 presents the evaluation of the proposed algorithm, in comparison with existing practices and theoretical works, while Section 5.5 analyze the interaction of our proposal with other scheduling goals. Section 5.6 provides an overview of closely related works, and Section 5.7 draws some conclusions and highlights future directions.

5.2 Background

A *MapReduce* system is a framework for distributed computation over large data sets that implements both the MapReduce programming model and an associated run-time system. It mimics the functional programming constructs *map* and *reduce* and enables the programmer to abstract from common distributed programming issues such as: load balancing, network performances and fault-tolerance. In spite of its simplicity, the MapReduce programming model turns out to effectively fit many problems encountered in the practice of processing large data sets although a preliminary decomposition of the problem into multiple MapReduce jobs is often needed [39, 84]. Typical applications are Web indexing, report generation, click-log file analysis, financial analysis, data mining, machine learning, bio-informatics and scientific simulations [48, 49].

The MapReduce programming model is based on the iteration over data-independent inputs where the required operations are: i) computation of key/value pairs from each piece of input (*map* phase); ii) grouping of all intermediate values by the key value; iii) reduction of each data group to a few computed values (*reduce* phase). Word counting is a toy example that considers a set of text documents as input and a list of the occurrences of each word as output, where the key/value pair is given by "word"/"counting" instances.

Actual implementations of proprietary [48, 49] and open-source [138] instances of a *MapReduce* system employ dedicated clusters of commodity machines. Each cluster is managed by a *master* server that is in charge of keeping track of all jobs while they are queued and processed in the distributed system. A *job-tracker* running on the master server schedules the received jobs and assigns their tasks on target *slave* servers. Each slave server runs a *task-tracker* that schedules the corresponding tasks, on a first-come/first-served strategy, consistently with the local computational resources and operating system policies. Due to the simplicity of the MapReduce programming model, a user will seldom submit a single job, since, the composition of more jobs in complex workloads (or applications) allows to take better advantage of the system. A MapReduce application is, in general, a Directed Acyclic Graph (DAG) where the nodes represent jobs and the arcs represent data dependences [39]. Therefore a job can only be executed after all of its predecessors have been completed.

Canonical solutions to the scheduling of a DAG solve a constrained optimization problem where the figure of merit is the expected latency of every job and the constraints are represented by the available resources.

A variant of this setting is to employ the minimization of resources as a figure of merit, and the maximum latency allowed for each job as a constraint. However, these strategies cannot be applied in the job-tracker, because they need a precise knowledge of the foreseen latency of each job as well as the available resources. The latency of a MapReduce job is not trivial to predict. This is due to both the heterogeneity of applications submitted by different users, and to the presence of straggled tasks and execution failures, which can change unpredictably the actual latency of the executed job [86]. In addition, the submission rate of the jobs in a Data Intensive Scalable Computing (DISC) cluster is quite low — on average, one job per 2-3 minutes [40, 86] — and thus the time to fill a queue of jobs to schedule is high. Given the aforementioned considerations, the scheduling strategy for the job-tracker of a MapReduce system should take into account the cluster workload variation over time. Therefore, *online* scheduling algorithms represent the prime choice.

Indeed, proprietary and open-source MapReduce systems adopt online scheduling strategies. Apache Hadoop [31] is an open-source Java implementation of *MapReduce*, originally designed to implement parallel processing in local networks, whose job-tracker employs a round-robin strategy (over the available resources) to assign the tasks in each job over the slave servers. A more accurate task assignment algorithm is proposed in [53], where the authors describe a flow-based algorithm aimed at minimizing the completion time of the considered job and show how such solution is near-optimal within an additive constant from the optimum solution obtained through the fully combinatorial exploration of task assignments.

We extend the abstract system model presented in [53], to effectively obtain a trade-off between job latency and throughput. Moreover, through taking into account a pre-existing workload, we better represent the challenges of an on-line task assignment.

5.3 The LABL Approach to Task Assignment

In this section, we introduce the main contribution of this work, a *Locality Aware Bounded Latency* (LABL) task assignment algorithm. We will now provide some preliminary concepts and definitions, followed by a description of the algorithm. We describe the formal properties of the LABL task assignment algorithm, and show that its running time complexity is linear whit respect to the size of the input job.

5.3.1 Preliminaries

We will first introduce some preliminary definitions, required to clearly describe our proposal.

Definition 5.1. A *job* is a set of *tasks*, $T = \{t_1, \ldots, t_m\}$. The tasks are mutually independent and do not have any control or data dependencies among them. Thus, the job can be fully parallelized.

In a *MapReduce* implementation, the tasks are partitioned between *map* and *reduce* operations. The *reduce* tasks must be scheduled after the *map* tasks have completed [138]. Without loss of generality, it is safe to model jobs as composed only of *reduce* tasks or only of *map* tasks. A job composed of both types of tasks is split in two homogeneous jobs for the purpose of the model, with the provision that the *reduce* job is scheduled only after the corresponding *map* job has completed. Note also that, in practice, the distribution of latencies of *reduce* tasks is remarkably similar to that of *map* tasks [86], so it is not necessary to keep track of *map* and *reduce* jobs separately.

Definition 5.2. A *cluster* is modelled as a set of homogeneous *servers*, $S = \{s_1, \ldots, s_n\}$, each of which is assumed to be able to execute a given task with the same execution time, provided that a copy of the corresponding data is locally accessible.

The locality of the data processed as the input of each task is crucial for the performance of the whole system. Indeed, the overall performance in terms of both job latency and total system workload largely depends on the initial data placement on the cluster.

Definition 5.3. Given a job T and a cluster S, a *data placement function* ρ specifies the subset of servers where the execution of a task t can be completed through accessing a local copy of the necessary data.

$$\rho : T \mapsto 2^S \wedge \forall t \in T, \rho(t) \subseteq S$$

The number of data copies available for a given task $t \in T$ is denoted as $|\rho(t)|$. A task t is denoted as *local* to a server s if $s \in \rho(t)$, and as *remote* otherwise.

As previously mentioned, the considered abstract model assumes a set of homogeneous tasks and a set of homogeneous servers, in such a way that all the tasks which data is locally available run in the same amount of time (w_{loc}) and all tasks running on servers where remote data accesses must be employed also exhibit the same execution time (w_{rem}).

The execution time experienced by the latter type of tasks depends on the total number of remote data accesses observed in the system. However, the additional overhead (with respect to the execution time of a task accessing data in place) does not incur in large variations when the network traffic of the system is in a steady state [53]. Therefore, the usual conservative assumption about the execution time experienced by tasks accessing remote data (fitting most of the practical environments) considers these execution times constant (over the entire set of tasks). In particular, the execution times are three times higher than the ones of tasks accessing data in place [53, 107].

Definition 5.4. Given a job T and a cluster S, an *assignment* corresponds to the execution of a number of tasks $\{t_1, \dots, \} \subseteq T$ on a single server $s \in S$, and is denoted as a pair $(s, \{t_1, \dots, \})$. A *Task Assignment*, $\mathcal{A}$, is a collection of pairs (s', T') with $s' \in S$, $T' \subseteq T$, such that every task in T and every server in S is present in one and only one assignment.

$$\mathcal{A} = \left\{ \begin{array}{rcl} (s', T') & : & s' \in S, T' \subseteq T \\ \forall s'' \in S, T'' \subseteq T \nexists (s'', T'') & : & s'' = s' \vee T'' = T' \end{array} \right.$$

The assignment of tasks to servers dynamically influences the subsequent assignment choices, due to the potential change of both network traffic and workload level of the cluster. The *job-tracker*, running on the *master* server, is the system actor in charge of orchestrating the workload distribution thus, it can dynamically evaluate the *load* of each server. Assuming w_{loc} and w_{rem} as the unitary task execution times for processing local and remote data, respectively, the evaluation of any server load is abstracted through the definition of the following function. We call the time w_{loc} a *unit of work*.

Definition 5.5. Let T be a job, S be a cluster, and $\mathcal{A}$ a given task assignment. The *load* of any server $s \in S$ is evaluated through a function, ϕ, which maps s to the numerical value of its current workload (measured in *units of work*). The workload of s in assignment $\mathcal{A}$ includes the set of tasks $\widehat{T} \subseteq T$, such that $(s, \widehat{T}) \in \mathcal{A}$. Then,

$$\phi(s) = \phi_s + w_{\text{loc}} |\widehat{T}_{\text{loc}}| + w_{\text{rem}} |\widehat{T}_{\text{rem}}|$$

where $\widehat{T}_{\text{loc}} = \{t \in \widehat{T} : s \in \rho(t)\}$ and $\widehat{T}_{\text{rem}} = \{t \in \widehat{T} : s \notin \rho(t)\}$ denote the sets of task that access data to be processed locally or remotely, respectively, while ϕ_s is a constant factor that takes into account the load due to the tasks that are already running on s before the assignment $(s, \widehat{T})$ is put into effect.

Note that, without loss of generality, we consider that at least one server s_0 has an initial workload $\phi_{s_0} = 0$, i.e. there is at least one free server. To understand the rationale of this choice, consider a load ϕ for a given cluster S, leading to an assignment $\mathcal{A}$. Now, consider a second load ϕ' such that $\forall s \in S, \phi'_s = \phi_s + 1$. The same assignment is generated under this second workload, except that the starting time of each task is increased by one unit of time. Thus, to provide a uniform scale for latency measurements, we normalize ϕ so that the condition $\exists s_0 \in S \mid \phi_{s_0} = 0$ holds.

5.3.2 Optimization Goals

Given a job T and a cluster S, the proposed task assignment strategy aims at achieving a trade-off between the *job latency* and the total *resource accounting* of the target cluster. The figures of merit used to evaluate the effectiveness of a task assignment algorithm `alg` and the resulting Task Assignment $\mathcal{A}$ are the following:

i) The *resource accounting* is defined as the total number $C_{\texttt{alg}}(T)$ of units of work consumed to execute the job:

$$C_{\texttt{alg}}(T) = \sum_{s \in S} (\phi(s) - \phi_s)$$

ii) The *latency* $l_{\texttt{alg}}(T)$ is defined as the maximum completion time for a task of the job, normalized to the minimum starting time for a task:

$$l_{\texttt{alg}}(T) = \max_{s \in S} \phi(s)$$

iii) The *throughput* is defined as the ratio $R_{\texttt{alg}}(T)$ between the number of tasks in the job and its resource accounting:

$$R_{\texttt{alg}}(T) = \frac{|T|}{C_{\texttt{alg}}(T)}$$

5.3.3 Lower Bounds for the Expected Job Latency

We start from the insight that it is possible to drive the online task assignment procedure taking as a reference a lower bound on the job latency. Such a reference allows the assignment procedure to start with

a predetermined minimum job latency limit, discarding unfeasible scenarios a-priori and taking into account remote assignments that would not be considered under lower latency limits.

Given a job T and a idle cluster S (i.e. $\forall s \in S, \phi_s = 0$), if each task can access the data to be processed on every server locally (i.e., $\forall t \in T, \rho(t) = S$), then a trivial lower bound for the job latency is given by $\lceil w_{\text{loc}}|T|/|S| \rceil$. Weakening these assumptions through removing either the hypothesis that each server is initially idle or the hypothesis of a uniform placement of data for each task, leads to solve two simpler problems prior to apply any task assignment operation. These problems are more formally stated as follows.

Problem 5.1. Let S be a cluster with initial workload defined by function $\phi(s) = \phi_s, \forall s \in S$, and T be a set of tasks that can locally access the data to be processed on any server $S : \forall t \in T \rho(t) = S$.

Considering the execution cost of each task as w_{loc} (that is, ignoring the impact of the data placement), a lower bound for the job latency can easily be computed a-priori as:

$$l^* = \left\lceil \frac{w_{\text{loc}}|T|}{|S|} + \frac{1}{|S|} \sum_{s \in S} \phi(s) \right\rceil$$

The straightforward solution of Problem 5.1 follows from considering each task as a local one since the data is assumed to be uniformly replicated on each server.

Problem 5.2. Let S be a cluster with initial workload defined by function $\phi(s) = \phi_s, \forall s \in S$, and T be a set of tasks whose data is replicated on servers according to a data placement function $\rho : T \mapsto 2^S$. Assuming a limit l for the expected job latency, the set S can be partitioned as $S = S_{\text{inf}}[l] \cup S_{\text{sup}}[l] \cup S_{\text{busy}}[l]$, where $S_{\text{sup}}[l] = \{s \in S | l - \phi_s \geq w_{\text{rem}}\}$ is the set of servers that can only execute local tasks within the latency limit l, $S_{\text{busy}}[l] = \{s \in S | l - \phi_s \leq 0\}$ is the set of servers that are busy with workload from previous jobs. Finally, the set of servers that cannot execute remote tasks within the latency limit l is $S_{\text{inf}}[l] = S \setminus (S_{\text{inf}}[l] \cup S_{\text{busy}}[l])$. The set of tasks T can also be partitioned as $T = T_{\text{loc}}[l] \cup T_{\text{rem}}[l]$, where $T_{\text{rem}}[l] = \{t \in T | \rho(t) \subseteq S_{\text{busy}}[l]\}$ is the set of tasks that can only be executed remotely within l and $T_{\text{loc}}[l] = T \setminus T_{\text{rem}}[l]$ is the set of tasks that can be run within l on servers with local access to data.

Considering the execution time of any task in T_{loc} as w_{loc} and the execution time of any task in T_{rem} as w_{rem} ($> w_{\text{loc}}$), a lower bound for the expected job latency is derived as:

$$l^{**} = \min_{l \geq 0} \begin{cases} \sum_{s \in S_{\text{sup}}} \left\lfloor \frac{l - \phi_s}{w_{\text{rem}}} \right\rfloor & \geq & |T_{\text{rem}}[l]| \\ \sum_{s \in S_{\text{sup}} \cup S_{\text{inf}}} (l - \phi_s) & \geq & a[l] \end{cases}$$

where $a[l]$ is the cost of the execution of the given job following an "ideal" assignment of both local and remote tasks within the latency limit l (in this way, the data placement function is employed only for partitioning the job in the local $T_{\text{loc}}[l]$ and remote $T_{\text{rem}}[l]$ task sets but not to solve assignment conflicts, if any):

$$a[l] = w_{\text{rem}}|T_{\text{rem}}[l]| + w_{\text{loc}}|T_{\text{loc}}[l]|$$

The first inequality states that the servers in S_{sup} can provide, as a whole, enough units of work to manage the execution of all remote tasks within the latency limit of l, while the second inequality constraints the available number of units of work on the entire cluster to be greater than the resource allocation needed to schedule each local task locally and each remote task remotely assuming no resource conflict. Therefore, the minimum among the aforementioned latency limits gives a lower bound l^{**} which guarantee a more accurate estimate with respect to the previous bound l^{*}, thus allowing to initialize our on-line assignment algorithm with a threshold that guarantee a faster convergence.

5.3.4 Task Assignment Algorithm

The LABL Task Assignment algorithm, reported in Figure 5.1, takes as input a job T, a cluster S and a lower bound l for the expected job latency that will be employed to drive the assignments computed as output. The initial value of the job latency limit l is equal to the lower bound l^{**}, computed as shown in the previous section.

The main loop of the algorithm iterates until all tasks are assigned to a server and is structured in three phases each of which acts on a different partition of the set of slave servers. At the beginning, the following subsets of servers and tasks are considered. S_{inf} includes all servers that can execute at least one local task within the limit l but not a remote one, while S_{sup} includes those servers that can execute at least one remote task within the limit l. Servers in the complementary set $S_{\text{busy}} = S \setminus (S_{\text{inf}} \cup S_{\text{sup}})$ will not be considered until the limit l for the job latency is increased, thus leading to consider them in S_{inf} or S_{sup} in subsequent iterations of the main loop. The job T is partitioned in two subsets: T_{loc} and T_{rem}, where T_{loc} includes any task that can be

Algorithm: TaskAssignment

Input: $S = \{s_1, \ldots, s_m\}$, set of servers
$T = \{t_1, \ldots, t_n\}$, set of tasks
l, initial server load limit

Output: $A = \{$
$(s, \widehat{T}) : s \in S, \widehat{T} \in \wp(T);$
$\forall (s', \widehat{T}'), (s'', \widehat{T}''), s' \neq s'' \wedge \widehat{T}' \cap \widehat{T}'' = \emptyset$
$\}$, set of assignments

// Place tasks on servers through trading off the job latency and
// data movement
1 $A \leftarrow \emptyset$
2 **while** $T \neq \emptyset$ **do**
3 $S_{\text{inf}} \leftarrow \{s \in S, l - w_{\text{rem}} < \phi(s) < l\}$
4 $S_{\text{sup}} \leftarrow \{s \in S, 0 \leq \phi(s) \leq l - w_{\text{rem}}\}$
5 $T_{\text{loc}} \leftarrow \{t \in T, \rho(t) \cap \{S_{\text{inf}} \cup S_{\text{sup}}\} \neq \emptyset\}$
6 $T_{\text{rem}} \leftarrow \{t \in T, \rho(t) \cap (S \setminus \{S_{\text{inf}} \cup S_{\text{sup}}\}) = \emptyset\}$
// Place most constrained tasks in T_{loc} on most loaded servers unable to
// execute a remote task while limiting their load under l
// (i.e. servers in S_{inf})
// $T_{\text{loc}} \cup T_{\text{rem}} = T$
7 $\widetilde{A} \leftarrow$ AssignCriticalTasks (S_{inf}, T_{loc}, l)
8 $A \leftarrow A \cup \widetilde{A}$
9 $T_{\text{loc}} \leftarrow T_{\text{loc}} \setminus$ ExtractAssignments ($\widetilde{A}$)
// Place remote tasks on servers s having a load such that
// $l - \phi(s) \geq w_{\text{rem}}$
10 $\widetilde{A} \leftarrow$ AssignRemoteTasks (S_{sup}, T_{rem}, l)
11 $A \leftarrow A \cup \widetilde{A}$
12 $T_{\text{rem}} \leftarrow T_{\text{rem}} \setminus$ ExtractAssignments ($\widetilde{A}$)
// Place tasks on less loaded servers storing the corresponding data
13 $T \leftarrow T_{\text{loc}} \cup T_{\text{rem}}$
14 $\widetilde{A} \leftarrow$ AssignToLessLoadedServers (S, T, l)
15 $T \leftarrow T \setminus$ ExtractAssignments ($\widetilde{A}$)
16 $l \leftarrow l + 1$
17 **return** A

Figure 5.1: Locality Aware & Bounded Latency (LABL) task assignment algorithm. The algorithm is composed by different phases, executed until all tasks have been assigned. See Figure 5.2, Figure 5.3, and Figure 5.4 for algorithms executed by AssignCriticalTasks, AssignRemoteTasks, and AssignToLessLoadedServers respectively

Algorithm: AssignCriticalTasks
Input: $S_{\text{inf}} = \{s_1, \ldots, s_m\}$, set of servers
$T_{\text{loc}} = \{t_1, \ldots, t_n\}$, set of tasks
l, server load limit
Output: $A = \{$
$(s, \widehat{T}) : s \in S, \widehat{T} \in \wp(T_{\text{loc}});$
$\forall (s', \widehat{T}'), (s'', \widehat{T}''), s' \neq s'' \wedge \widehat{T}' \cap \widehat{T}'' = \emptyset$
$\}$, set of assignments

// Place most constrained tasks in on most loaded servers unable to
// execute a remote task while limiting their load under l
1 $\widehat{A} \leftarrow \emptyset$
2 **while** $S_{\text{inf}} \neq \emptyset$ **do**
// Get $s \in S$ s.t. $\phi(s) > \phi(s_i), \forall s_i \in S_{\text{inf}}, s_i \neq s$
3 $s \leftarrow$ ExtractMostLoadedSrv (S_{inf})
4 $\widetilde{T} \leftarrow \rho^{-1}(s)$ // Set containing tasks working on s local data
5 $\widehat{T} \leftarrow \emptyset$ // Set of tasks foreseen to be assigned to s
6 **while** $\widetilde{T} \neq \emptyset$ **and** $\phi(s) \leq l - w_{\text{loc}}$ **do**
// Get $t \in \widetilde{T}$ s.t. $|\rho(t)| < |\rho(t_i)|, \forall t_i \in \widetilde{T}, t_i \neq t$
7 $t \leftarrow$ ExtractMostConstrainedTask $(\widetilde{T})$
8 $\widehat{T} \leftarrow \widehat{T} \cup \{t\}$
9 $\widehat{A} \leftarrow \widehat{A} \cup \{(s, \widehat{T})\}$
10 **return** $\widehat{A}$

Figure 5.2: Critical tasks assignment. To maximize the usage of available resources, servers who cannot execute tasks remotely are loaded with locally executable tasks. Tasks selection is done according to the number of servers who can access data locally, in order to assign most constrained tasks first

executed on at least one server in $S_{\text{inf}} \cup S_{\text{sup}}$ and T_{rem} includes any task that can only be executed remotely before the limit l.

The body of the main loop is divided in three phases. In the first phase (Figure 5.2), we assign as many tasks as possible from T_{loc} to servers in S_{inf}, without exceeding the limit l. The tasks from T_{loc} are selected in ascending order of $|\rho(t)|$ (i.e., ranked by the number of servers where they can access data locally), so as to assign first those tasks that can only be executed on few servers, and are therefore more likely to cause violations of the target latency l. This is due to the fact that the initial value of l is l^{**}, which has been computed without taking into account the effect of many tasks having data on a small group of servers.

In the second phase (Figure 5.3), we assign tasks from T_{rem} to servers

Algorithm: AssignRemoteTasks

Input: $S_{\text{sup}} = \{s_1, \ldots, s_m\}$, set of servers
$T_{\text{rem}} = \{t_1, \ldots, t_n\}$, set of tasks
l, server load limit

Output: $A = \{$
$(s, \widehat{T}) : s \in S, \widehat{T} \in \wp(T_{\text{rem}});$
$\forall (s', \widehat{T}'), (s'', \widehat{T}''), s' \neq s'' \wedge \widehat{T}' \cap \widehat{T}'' = \emptyset$
$\}$, set of assignments

// Place remote tasks on servers having a load such that they
// can execute within the target latency

```
1  Â ← ∅
2  if ConsiderRemoteAssignments (l) = true then
3    S'_sup ← ∅
4    while T_rem ≠ ∅ do
5      t ← ExtractTask (T_rem)
6      s ← ExtractSrv (S_sup)
7      T̂ ← ExtractAssignment (Â, s)
8      T̂ ← T̂ ∪ {t}
9      Â ← Â ∪ {(s, T̂)}
10     if φ(s) ≤ l − w_rem then
11       S'_sup ← S'_sup ∪ {s}
12     if S_sup = ∅ then
13       S_sup ← S'_sup
14       S'_sup ← ∅
15 return Â
```

Figure 5.3: Remote tasks assignment. In order to execute tasks within the estimated deadline l, remote tasks are assigned to servers whose load allows executing at least one of them. This phase is optional. Indeed, it is executed only when the current value of l is below the threshold function ConsiderRemoteAssignments

Algorithm: AssignToLessLoadedServers
Input: $S = \{s_1, \ldots, s_m\}$, set of servers
$T = \{t_1, \ldots, t_n\}$, set of tasks
l, initial server load limit
Output: $A = \{$
$(s, \widehat{T}) : s \in S, \widehat{T} \in \wp(T);$
$\forall (s', \widehat{T}'), (s'', \widehat{T}''), s' \neq s'' \wedge \widehat{T}' \cap \widehat{T}'' = \emptyset$
$\}$, set of assignments

// Place tasks on less loaded servers storing the corresponding data
1 $\widehat{A} \leftarrow \emptyset$
2 **while** $T \neq \emptyset$ **do**
3 $t \leftarrow$ ExtractTask (T)
4 $s \leftarrow$ ExtractLeastLoadedSrv $(\rho(t))$
5 **if** $\phi(s) \leq l - w_{\text{loc}}$ **then**
6 $\widehat{T} \leftarrow$ ExtractAssignment $(\widehat{A}, s)$
7 $\widehat{T} \leftarrow \widehat{T} \cup \{t\}$
8 $\widehat{A} \leftarrow \widehat{A} \cup \{(s, \widehat{T})\}$
9 **return** $\widehat{A}$

Figure 5.4: Less loaded servers assignment. The last phase of the algorithm is to distribute remaining tasks across available servers. Servers are considered according to their load, while tasks are assigned only if the latency bound given by l is respected. If some task is left unassigned, the three phases will be repeated, considering an augmented value of l

in S_{sup}, without exceeding the limit l. During the first iteration of the main loop, all tasks from T_{rem} might be assigned, because the limit l is initially set to l^{**}, which guarantees that all tasks that need to be executed remotely can be completed within l^{**}.

In the third phase (Figure 5.4), we assign as many tasks as possible from T_{loc} to servers in S_{sup}, without exceeding the limit l. Finally, if some tasks are still unassigned, the algorithm increases the limit l by one unit, recomputes the four subsets (T_{loc}, T_{rem}, S_{inf}, S_{sup}) and iterates the three phases.

Note that the second phase forces the assignment of as many remote tasks as possible, employing time that could be usefully exploited by other jobs in return for a potentially very low latency gain. Thus, the algorithm triggers the execution of the second phase by means of a *threshold function* (ConsiderRemoteAssignments in Figure 5.3, Line 2) that is *true* until a given latency limit is reached, and *false* thereafter.

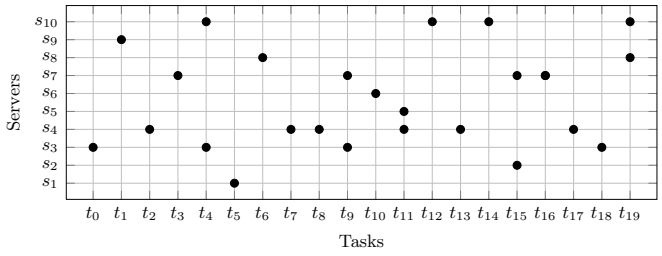


Figure 5.5: An example of data placement. A marker at coordinate (t_i, s_j) means that data accessed by t_i during its execution is stored on server s_j

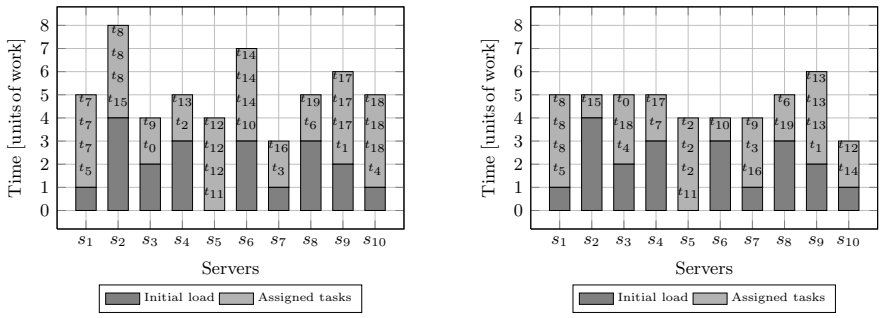
5.3.5 Case Study

To understand the behavior of the LABL algorithm, we compare it to the locality-aware round-robin [138] and flow-based algorithms [53], using a limited number of servers, $|S| = 10$, and tasks, $|T| = 20$. The task execution times are set at $w_{\text{loc}} = 1$, $w_{\text{rem}} = 3$. Figure 5.5 reports the considered data placement, with a maximum data replication factor of 2. Figure 5.6 reports assignments generated by the round-robin algorithm [138] and the flow-based algorithm [53], while Figure 5.7 shows assignments generated by the LABL algorithm, when the execution of the second phase is stopped after the first iteration.

The round-robin algorithm cycles through the list of servers in a predetermined arbitrary order until all tasks have been assigned (in the example, starting from s_1, then s_2, s_3, etc.). At each step, a task is assigned to a server. The algorithm tries to exploit the data placement by assigning a local task to the current server. If this is not possible, a remote task is assigned. The greedy choices of the round-robin algorithm results in a final assignment (see Figure 5.6(a)) with high job latency and high resource consumption ($l_{\texttt{rr}} = 8$, $C_{\texttt{rr}}(T) = 32$).

The approach reported in [53] improves the round-robin strategy and describes an algorithm that allows to choose the minimum latency assignment among a list of $|T|$ possibilities. Each assignment is computed through a flow-based approach to maximize the assignment of local tasks (while limiting the load of the corresponding servers under a temporary threshold) followed by a greedy strategy necessary to complete the assignment of remote tasks. Figure 5.6(b) shows the assignment resulting from the aforementioned strategy ($l_{\texttt{flow}} = 6$). We note that the greedy choice, applied to assign the remote tasks, can often lead to resource consumption higher than the minimal one: $C_{\texttt{flow}}(T) = 26 > 20$.

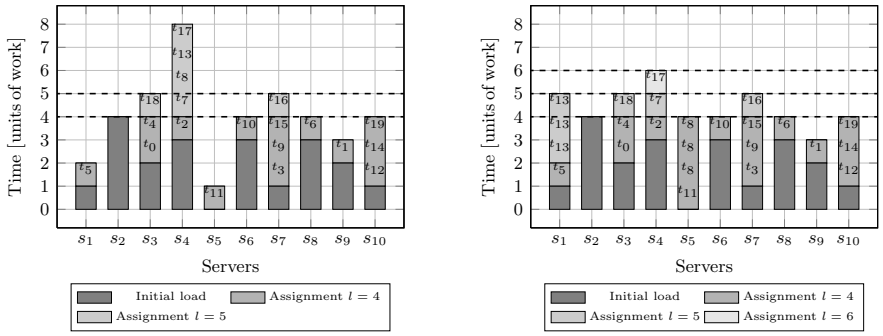
Figure 5.7 depicts the assignments computed by the LABL algorithm



(a) Round-robin: $C_{\texttt{rr}}(T) = 32$ units

(b) Flow-based: $C_{\texttt{flow}}(T) = 26$ units

Figure 5.6: Round-robin (5.6(a)) and flow-based (5.6(b)) assignments



(a) $C_{\texttt{LABL}}(T) = 20$ units. Remote assignment performed considering latency threshold $l = 4$

(b) $C_{\texttt{LABL}}(T) = 24$ units. Remote assignments performed considering latency threshold $l \in \{4, 5\}$

Figure 5.7: Locality Aware & Bounded Latency task assignments

when taking as input an initial job latency limit $l = 4$. The algorithm exhibits different behaviors in terms of total job latency and minimization of resource allocation depending on the configuration of the *threshold function* (see ConsiderRemoteAssignments in Figure 5.3, Line 2) that stops the execution of the second phase of the algorithm from a specified iteration on. Figure 5.7(a), shows the assignments obtained when the second phase is executed only at the first iteration.

Note that this has no effect on the final assignment since, at the first iteration, there is no tasks that needs to access data remotely. Indeed, the initial servers load specified in Figure 5.7(a) suggests that only tasks local to server s_2 may be considered for remote assignment. The data placement function specifies that t_{15} is the only task that can be assigned on s_2, however t_{15} is also local to server s_7. Thus, t_{15} has to be assigned

on s_7. The final assignment in Figure 5.7(a) uses resources sparingly ($C_{\texttt{LABL}}(T) = 20$, equal to the minimum), at the cost of an increased latency ($l_{\texttt{LABL}} = 8$).

To decrease latencies, it is necessary to consider the explicit handling of remote tasks up to the second iteration (Figure 5.7(b)). This allows to assign tasks t_8 and t_{13} remotely, contributing to lower the overall latency, at the cost of an increased resource usage. With respect to the assignment found by the flow-based algorithm, we achieve the best possible combination of job latency $l_{\texttt{LABL}} = 6$ and resource usage $C_{\texttt{LABL}}(T) = 24$.

5.3.6 Formal Properties of the LABL Task Assignment

In this section, we analyze the properties the LABL Task Assignment algorithm. We first prove that the algorithm can be configured by manipulating the ConsiderRemoteAssignments threshold function to achieve strong properties on load balance and resource usage. Subsequently, we analyze the computational complexity of the LABL algorithm.

Theorem 5.1. *Under the condition that* ConsiderRemoteAssignments *is* true *for all iterations of the main loop, the LABL Task Assignment algorithm produces an assignment* $\mathcal{A}_{LABL}$ *with*

$$\max_{s \in S} \phi(s) \leq \min_{s \in S} \phi(s) + w_{\text{rem}}$$

Proof. Let $s_{\max}$ be one of the servers such that the latency of the computed assignment is $l_{\texttt{LALB}} = \phi(s_{\max})$ and $s_{\min}$ be another server such that the execution of the tasks on it makes its final completion time $\phi(s_{\min})$ equal to the minimum latency among the servers in S. The proof will be developed through a *reductio ad absurdum*.

Assume that $\phi(s_{\max}) > \phi(s_{\min}) + w_{\text{rem}}$ holds at the end of the LABL algorithm execution, and that the latency of the computed assignment is $l_{\texttt{LABL}} = l_{\texttt{out}}$. Such an hypothesis implies that in the last-but-one iteration of the outer loop of the LABL algorithm, there was a number n of tasks that could not be assigned within the latency limit $l = l_{\texttt{out}} - 1$. In the case $n = 1$, this task would have been assigned to the server $s_{\min}$ in the second phase of the algorithm, as the hypothesis guarantees enough resources for the remote execution of it. This contradicts the initial assumption as the aforementioned last iteration would not have occurred, and therefore the latency of the computed assignment would have been $l_{\texttt{LABL}} = l_{\texttt{out}} - 1$.

In case $n > 1$, each task can be sequentially assigned for remote execution to a server, starting from the one having workload equal to $\phi(s_{\min})$, as long as the number of tasks and the number of servers satisfying the condition $\phi(s) + w_{\text{rem}} \leq l_{\texttt{out}} - 1$ allows the assignments. If all tasks are assigned, then the last iteration would not have occurred, thus having the same conditions of the former case. Otherwise, the remaining tasks must be assigned at the next iteration when $l = l_{\texttt{out}}$ as the servers in the last-but-one iteration could have included only tasks requesting an execution time in $[w_{\text{loc}}, w_{\text{rem}} - 1]$ which is not obviously the case. In the last iteration there would have been only servers that could satisfy assignments of tasks with an execution time ranging from w_{loc} to w_{rem}. Therefore the difference between the maximum and the minimum workload would be $\phi(s_{\max}) - \phi(s_{\min}) \leq w_{\text{rem}}$, that contradicts the hypothesis. □

Corollary 5.1. *If Theorem 5.1 holds and the server $s_{min} \in S$ with minimum workload satisfies the condition $\phi(s_{min}) \leq l^{**}$, then the optimal latency l^{opt} for the given assignment problem is bounded as follows:*

$$l_{LABL} - w_{\text{rem}} \leq l^{opt} \leq l_{LABL}$$

Proof. The lower bound given by l^{**} is lesser than or equal to $l^{\texttt{opt}}$ by definition, while $l^{\texttt{opt}}$ is, in turn, lesser than or equal to the latency limit computed by the LABL algorithm: $l^{**} \leq l^{\texttt{opt}} \leq l_{\texttt{LABL}}$. Now, if Theorem 5.1 holds, then $l_{LABL} = \phi(s_{\max})$ and $\phi(s_{\min}) \geq l_{\texttt{LABL}} - w_{\text{rem}}$. Therefore, noting that l^{**} must be greater than or equal to $\phi(s_{min})$, leads to the thesis. □

Theorem 5.2. *The LABL Task Assignment algorithm, under the condition that* ConsiderRemoteAssignments *is* false *for all values of $l > l^{**}$, produces an assignment $\mathcal{A}_{LABL}$ with a total resource usage*

$$C_{LABL}(T) \leq l^{**} \times |S| - \sum_{s \in S} \phi_s$$

Proof. If ConsiderRemoteAssignments is false for all l except l^{**}, the second phase of the LABL algorithm is executed only once, that is the assignment of remote tasks is performed only in the first iteration (i.e., when $l = l^{**}$).

If all the tasks are assigned in the first iteration (that is, the algorithm computes a final latency $l_{\texttt{out}} = l^{**}$) then the resource allocation in terms of units of work is due to the servers in $S_{\text{sup}} \cup S_{\text{inf}} = S \backslash S_{\text{busy}}$, as in S_{busy} there are only servers with a workload that doesn't allow to cope with either local or remote tasks. Therefore the following relation holds:

$$\sum_{s \in S_{\text{sup}} \cup S_{\text{inf}}} (l^{**} - \phi_s) \quad \geq \quad \sum_{s \in S} (l^{**} - \phi_s)$$

The inequality right side ($\sum_{s \in S} (l^{**} - \phi_s) = l^{**} \times |S| - \sum_{s \in S} \phi_s$) is always smaller than the left one ($C_{\texttt{LABL}}$), as the workload of servers in S_{busy} is by definition greater than or equal l^{**}.

If the LABL assignment algorithm terminates with $l_{\texttt{out}} > l^{**}$, then through remembering that the latency limit given by l^{**} guarantees (by definition) that the whole cluster S can allocate all the remote tasks (see the first condition in the definition of l^{**} in Section 5.3.3), and following the theorem hypothesis the assignment of tasks in the first and third phase of the algorithm will proceed through allocating the tasks locally, it is easy to infer that the whole number of units of work actually spent by the cluster ($C_{\texttt{LABL}}$), at the end of the computation, will not be greater than $l^{**} \times |S| - \sum_{s \in S} \phi_s$. □

Theorem 5.3. *The LABL Task Assignment algorithm operates in time*

$$\mathcal{O}\left(\log |T| \times |T| \times \max_{t \in T} |\rho(t)|\right)$$

where $|T|$ is the number of tasks and $\max_{t \in T} (|\rho(t)|)$ is the maximum number of data copies available for a task.

Proof. We represent $\rho(t)$ as adjacency lists sorted by server load and $\rho^{-1}(s)$ as adjacency lists sorted by $|\rho(t)|$. The sorting of subsets of T can be performed employing a counting sort algorithm, and has therefore $\mathcal{O}(|T| + \max_{t \in T} (|\rho(t)|))$ complexity, since the number of keys is at most $\max_{t \in T} (|\rho(t)|)$. The sorting of subsets of S can also be performed employing a counting sort algorithm, and therefore its complexity is $\mathcal{O}(|S| + \max_{s \in S} (\phi(s)))$, since there are at most $\max_{s \in S} (\phi(s))$ keys. Note that the maximum values of $|\rho(t)|$ and $\phi(s)$ are two orders of magnitude smaller than $|T|$ and $|S|$ in real world cases, so using counting sort or other distribution sort algorithms is a reasonable choice. In particular, $\phi(s) \leq \max\{\phi_s, l^{**}\}$ initially, and $\phi(s) \leq \max\{\phi_s, l\}$ in successive iterations.

Computing the four sets S_{inf}, S_{sup}, T_{loc} and T_{rem} amounts to a single scan of S and T. Since in general $|S| < |T|$, the construction is overall $\mathcal{O}(|T|)$. The first phase scans the entire S_{inf}. At most w_{rem} tasks are assigned for each $s \in S_{\text{inf}}$, since doing otherwise would lead to violating the latency bound. The complexity of this phase is therefore $\mathcal{O}(|S|)$. The second phase scans the entire T_{rem}, and assigns all tasks to the least loaded servers in a round robin way. The complexity of this

phase is straightforward, as it performs $\mathcal{O}(|T_{\text{rem}}|)$ operations, which is also $\mathcal{O}(|T|)$. While the complexity of the third phase, as explained in Figure 5.1 is $\mathcal{O}(|T|)$, it is possible to implement it by iterating on the servers in S_{sup} and assigning as many task to each server as it can handle within the latency bound. This leads to a complexity of $\mathcal{O}(|S|)$.

Overall, the complexity of each loop iteration is thus bounded by $\mathcal{O}(|T| + \max_{t \in T} |\rho(t)|) + \mathcal{O}(|S| + \max_{s \in S} \phi(s))$. Since we increase l by one at each iteration, the number of iterations of the main loop is given by $l_{\texttt{LABL}} - l^{**}$, where $l_{\texttt{LABL}}$ is the latency of the assignment. Note that, even if we allocated every task remotely, $l_{\texttt{LABL}}$ would be limited by

$$l_{\texttt{LABL}} \leq \frac{w_{\text{rem}}|T| + \sum_{s \in S} \phi_s}{|S|}$$

Considering that l^{**} is at most equal to $(w_{\text{loc}}|T| + \sum_{s \in S} \phi_s)/|S|$, it follows that $l_{\texttt{LABL}} - l^{**} \leq (w_{\text{rem}} - w_{\text{loc}})|T|/|S|$. In general, it can be assumed that $|T| \simeq c|S|$, where c is a small factor typically ranging from 2 up to 10 [48, 86], therefore the outer loop is executed only a fixed number of times.

However, we ensure this by means of the threshold limit of l imposed by ConsiderRemoteAssignments. Thereafter, we perform a reduced loop including only the first and third phases. This reduced loop, per se, has a complexity $\mathcal{O}\left(|T|^2\right)$, but it can be usefully restructured whit respect to the general presentation to reduce the complexity. Specifically, since we are now only assigning tasks t to servers in $\rho(t)$, we can work as follows: for each $s \in S$, compute a set $R_s = \{t \in \rho^{-1}(s) \text{ if } t \in T\}$, and sort each set by $|\rho(t)|$.

We now iterate over the servers $s \in S$ in a round-robin way, removing one element of R_s at each iteration and assigning it to s if it has not been already assigned. This guarantees completion in

$$\mathcal{O}\left(\log|T| \times \sum_{s \in S} |R_s|\right) = \mathcal{O}\left(\log|T| \times |T| \times \max_{t \in T} |\rho(t)|\right)$$

which becomes the most computationally intensive phase of the algorithm. □

5.4 Simulation Results

We conducted an experimental campaign to compare the behavior of the LABL Task Assignment with the round-robin and flow-based algorithms. We employed as a starting point a real-world configuration from [48],

which provides statistical data on the execution of MapReduce jobs at Google during an entire month.

The experiments are conducted in a simulation environment, scheduling one job on a set of servers having an existing workload. This is done to simulate the online scheduling process: given the mean inter-arrival time of 2-3 minutes reported in [40, 86], the job tracker will have completed the scheduling process of the job before a second one arrives. On the other hand, due to the long computation times, previously scheduled jobs will still be active while the new one is being scheduled.

The simulation assumes tasks to require the same time w_{loc} to be executed on any server storing the necessary data. Since the time w_{loc} also represents a *unit of work*, we will consider $w_{\text{loc}} = 1$ in all experiments. Whenever a task is assigned to a server that does not have the required data, the data must be fetched, leading the execution time to increase to w_{rem}. We set $w_{\text{rem}} = 3$ in all experiments, following the same approach as [53].

We explore a configuration space considering a number of servers $|S|$ ranging from 1600 up to 2000, and a number of tasks $|T|$ between 3200 and 3500, though we will only show subsets of the overall configuration space in some experiments for the sake of clarity. The data placement is randomly determined such that $|\rho(t)|$ is in the range $[1, \rho_{\max}]$ for all tasks, where $\rho_{\max}$ is a parameter fixed at 4 in all experiments, except when evaluating the sensitivity of the algorithms to the replication factor. In all the experiments, the initial load is randomly assigned, within the range $[0, 5]$. In all cases, the reported data has been obtained as the average of the results gathered from 30 runs of the same experiment.

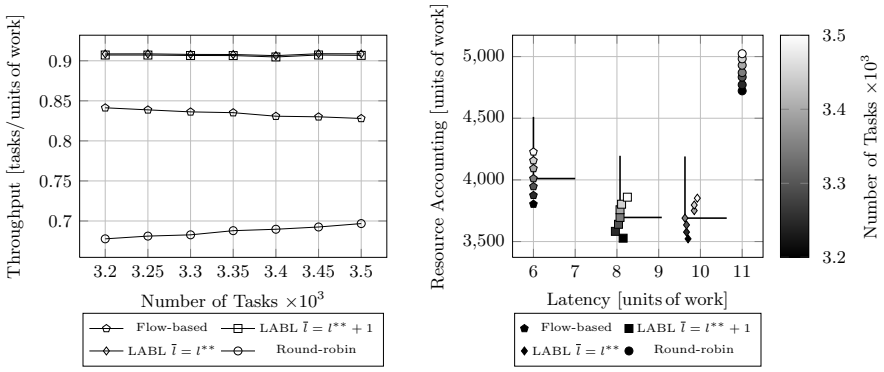
5.4.1 Performance Overview

The experiment reported in Figure 5.8 compares the effectiveness of the LABL Task Assignment with both the round-robin and flow-based algorithms, in terms of throughput, resource accounting and latency. We explore a configuration space with $|S| = 2000$, $|T| = \{3200, \ldots, 3500\}$.

Data for the LABL algorithm are reported for configurations with threshold latency $\bar{l}$ set to l^{**} and $l^{**} + 1$.

Figure 5.8(a) shows the throughput achieved by the three algorithms. The LABL algorithm, in both versions, yields a better throughput, i.e. the task assignment is able to consistently save resources, leaving more server time for other jobs.

Figure 5.8(b) reports in a scatter-plot the latency and resource consumption obtained by the three algorithms on the 2000 servers cluster, showing increasing number of tasks in the job by lighter shades. Fig-



(a) Throughput

(b) Pareto efficiency

Figure 5.8: Performance of analyzed algorithms

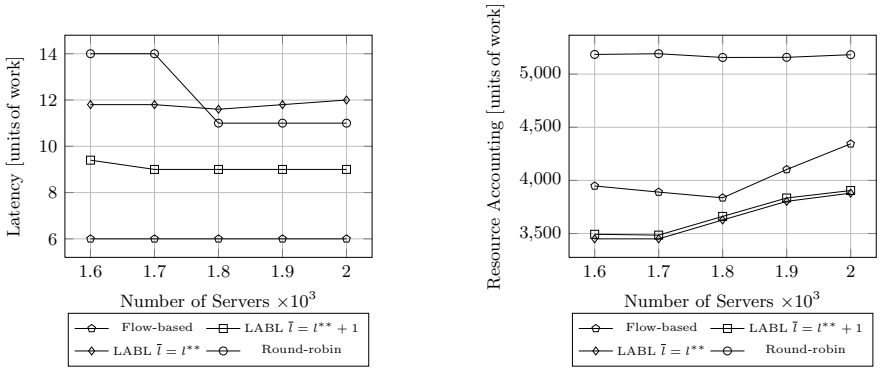
ure 5.8(b) shows that the flow-based algorithm consistently obtains optimal latencies, while the LABL algorithm reduces resource usage. The LABL algorithm and the flow-based algorithm produce solutions that are Pareto-optimal, while the round-robin algorithm produces solutions that are Pareto-dominated by all the others.

On the overall, the flow-based and LABL algorithms produce solutions of interest respectively to optimize latency and resource usage. However, the flow-based algorithm has a higher computational complexity, $\mathcal{O}(|T|^2 \times |S|)$ [53], making the LABL solution more attractive.

5.4.2 Scalability

The experiment reported in Figure 5.9 evaluates the robustness of the four algorithms to changes in the availability of servers. Given a set of tasks T, $|T| = 3450$, a data placement, and an initial workload, we progressively increase the number of servers that are available for scheduling from a minimum of $|S| = 1600$ to a maximum $|S| = 2000$.

A desirable property for the scheduling algorithm is that the number of available servers has only limited impact on the latency – assuming there are enough servers to actually execute the job. Figure 5.9(a) shows that only the round-robin algorithm is significantly impacted by the change in server availability. This is because the round-robin algorithm makes greedy choices, which easily prove suboptimal. The other three algorithms behave in a more graceful way, as their greedy choices are less aggressive – all four algorithms have greedy components within their heuristics, to limit the complexity, but the greedy component is dominant only in the round-robin algorithm. The LABL algorithm produces



(a) Latency

(b) Resource accounting

Figure 5.9: Scalability of the analyzed algorithms

Task Assignments with higher latencies than the flow-based algorithm. This is expected since, as shown in Section 5.4.1, the LABL algorithm trades off latency to save resources.

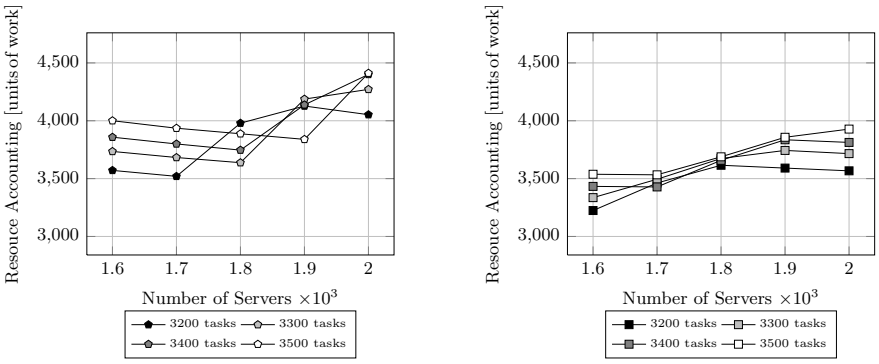
Figure 5.9(b) shows the impact of server availability on the resource usage. The impact is minimal on the round-robin algorithm, while the other three algorithms all tend to consume more resources when these are available, by placing remote tasks on free servers in an attempt to reduce latency. However, the LABL algorithm, in both versions, always outperforms the flow-based algorithm, thanks to its greater focus on reducing resource usage.

5.4.3 Sensitivity Analysis

The experiments reported in Figure 5.10 and Figure 5.11 evaluate the sensitivity of resource usage to, on one hand, the number of tasks to execute and the number of available servers, and, on the other hand, the replication factor, i.e. the average number of copies of the data accessed by a task.

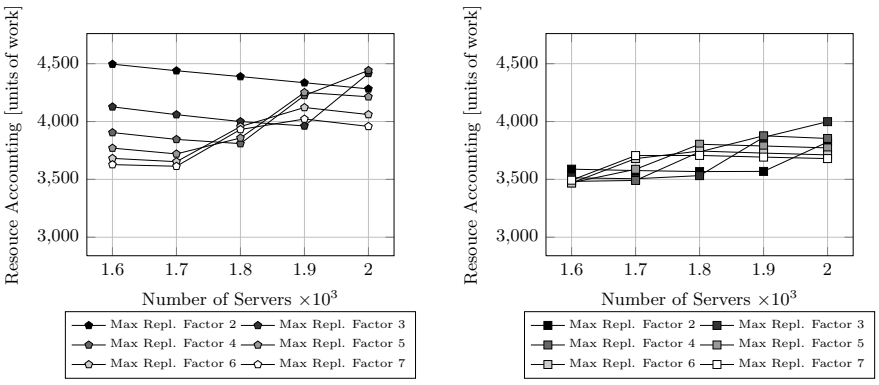
In the first case, only the resource accounting for the flow-based (Figure 5.10(a)) and LABL algorithm with $\bar{l} = l^{**} + 1$ (Figure 5.10(b)) are shown, as these algorithms have proven to be the most effective ones (see Figure 5.9). Figure 5.10 depicts a family of curves representing resource accounting as a function of the number of available servers $|S| = \{1600, \ldots, 2000\}$, considering the number of tasks to assign $|T| = \{3200, \ldots, 3500\}$ as a parameter.

As expected, the LABL algorithm consumes less resources. The results also show that the behavior of the LABL algorithm is much more



(a) Flow-based (b) LABL $\bar{l} = l^{**} + 1$

Figure 5.10: Resource awareness of analyzed algorithms



(a) Flow-based (b) LABL $\bar{l} = l^{**} + 1$

Figure 5.11: Replication factor sensitivity of analyzed algorithms

stable. Moreover, the flow-based algorithm is characterized by a higher resource usage when scheduling more tasks. Focusing on the replication factor, Figure 5.11 shows only the resource accounting employed by the flow-based and LABL algorithm (with $\bar{l} = l^{**} + 1$), as a function of the cluster size. The round-robin strategy is not considered since it consistently employs a higher number of resources (see Figure 5.9(b)). We vary the maximum replication factor $\rho_{\max}$ from 2 to 7, so that the average replication factor ranges in $[1.5, 4]$. Thus, the generated data placements have $|\rho(t)|$ uniformly distributed in the range $[1, \rho_{\max}]$ for all tasks.

The results show that the LABL algorithm is less sensitive to the replication factor than the flow-based one. The flow-based algorithm takes greater advantage from the increased locality given by the presence of more replicas of each data item, but the LABL algorithm is still able to achieve a lower resource usage. Note that a higher replication factor does impact on the overall costs — keeping up to date copies of the data across the network is bound to have a significant communication cost, so the ability to achieve good resource utilization with a low replication factor is a strong asset of the LABL algorithm.

5.5 Discussion

We will now discuss the interactions of the LABL algorithm with other scheduling goals such as fairness and adaptivity, as well as potential optimizations.

Scheduling for Fairness The *fairness* property is often desirable in large-scale clusters that are accessed by multiple users. That is, the applications submitted by any user should not be delayed indefinitely. Online scheduling strategies, such as the LABL algorithm, can be integrated into higher level policies aimed at providing such fairness guarantees, that is, at user-application scheduling level rather than at task-scheduling. Indeed, the LABL algorithm could effectively replace the round-robin algorithm that is used as the task assignment component of the *Hadoop fair scheduler* [53, 138].

Scheduling Jobs from Multiple Applications It is possible that, for a given job, some servers of the cluster have no copies of the required data for any of its tasks – or a set of servers $S' \subset S$ has only copies of data needed for a set of tasks $T' \subset T$, but $|T'| < |S'|$, leaving $|S'| - |T'|$ servers idle. In this case, the servers cannot be used to run a local

task, either leading to execution delays, if they are used to run a remote task, or to an under-utilization of resources. To further improve resource utilization and throughput, it is possible to schedule jobs from multiple applications at the same time, as these are likely to use different data sets. It is worth noting, however, that scheduling multiple jobs increases the throughput at a cost in latency.

The LABL scheduling algorithm, however, can easily handle the schedule of sets of tasks belonging to different jobs coming from independent applications, through simply merging the two sets. The key issue is selecting jobs that map on data held in different servers of the cluster, so as to allow servers that cannot run tasks locally for one job to be used for the other job.

Adaptive Scheduling A latency-aware scheduling is more attractive when the cluster is under-utilized, as it allows to minimize application latency, providing a better response time to the user. On the other hand, a resource-aware scheduling becomes increasingly important as the cluster utilization grows. Indeed, in a cluster under a heavy workload, a scheduling policy that favors latency may easily lead to low availability for other jobs. A common solution is to artificially limit the amount of resources that a single job can take. The LABL algorithm does that, by construction, optimizing the resource accounting of the scheduled job, while still providing a strong latency limit. Thus, it adapts better to workload variations, as shown in Section 5.4.3.

5.6 Related Work

The MapReduce programming model has been formalized in a number of ways. In [84] MapReduce computations have been compared to the PRAM model, focusing on analyzing how PRAM algorithms can be expressed using MapReduce.

Among the studies on task assignment, in [107] the authors focus on allocating tasks of multiple jobs in both on-line and off-line scenarios, providing a generalization of the Flexible-Flow Shop problem. However, the authors do not take into account the impact of data placement, which is critical due to the size of the exchanged data. In [53] the Hadoop round-robin based task allocator is compared with a flow-based task allocator, showing that careful consideration of data placement allows to limit job latency. An in-depth comparison with both algorithms is provided in Section 5.4. Job latency reduction has been tackled in [153] considering a production-quality scenario, showing how careful job speculation

helps on limiting the latency penalty introduced by straggled tasks (i.e., remotely executed tasks on the critical path), at the cost of an increased resource consumption. This technique, while applicable to all tasks, is more effective on *reduce* tasks, since map tasks are much less likely to be straggled.

In a typical MapReduce implementation, the set of available resources is equally exposed to all jobs. In [120], on the other hand, a different processing resources are exposed to each job depending on its workload profile in terms of CPU, disk and memory usage. Thus, a task tracker can maximize the use of its resources through executing tasks from jobs with different profiles. This scheme can be easily combined with our own, since in our approach the set of resources is an input parameter, whilst the key aspect of [120] is the definition of the resource set for each job profile.

In [145], a framework to estimate the latency of a MapReduce job as a function of the employed resources is introduced. The scheme is based on a job profile obtained through the execution of the same job on a smaller data set. This work, while not directly related to our own, could be adapted to provide stronger latency bounds for task assignment. This solution, however, would incur in the cost of job profiling.

In [149], FLEX, a scheduler for MapReduce systems, is proposed as a replacement for the Hadoop fair scheduling algorithm. With respect to our work, FLEX does not take into account data locality, and works on multiple jobs at the same time in an epoch-based scheme. Similarly, in [154] multiple jobs are managed, aiming at fairness and data locality, but with no latency guarantees.

The task assignment problem is common to all Data-Intensive Scalable Computing schemes. However, the solutions need to be tailored to the specific setup: e.g., [115, 117] deal with *cloud*-based MapReduce services, which rely on a heavy use of virtualization techniques. Virtualization is not attractive for every Data-Intensive Scalable Computing scenario, due to the need to spawn new virtual machines at high frequency — job completion times follow a long tailed distribution, with 80% of the successful jobs completing within 6 minutes, as shown in [86] for a 10-month timeframe on a production Yahoo! Hadoop cluster. In [81], on the other hand, a typical cluster of commodity machines is used to run tasks with dependencies, leading to different problems, such as the need to keep dependent tasks on near machines to minimize communications.

5.7 Concluding Remarks

We have presented an algorithm for assigning the tasks of a job to servers in a MapReduce cluster. The proposed algorithm balances the trade-off between latency and resource consumption. A comparison is drawn with both the locality-aware round-robin [138] and the flow-based algorithm [53], substantially improving the resource accounting while still providing a limited increase in latency. Since resource accounting is a proxy of energy consumption, the proposed algorithm can be effectively employed to fine tune the Quality of Service towards green solutions. Simulation results support the insight that a practical implementation would benefit from the proposed approach.

Future works include integrating the LABL task assignment algorithm within a higher level job scheduling framework, which would also manage fault tolerance issues. In addition, as a further refinement of the proposed technique, the cluster interconnect topology will be taken into account to model the remote execution time.

6 Towards Runtime Optimization of Parallel Applications

This chapter describes a dynamic and lightweight compiler able to guide the execution of parallel programs at runtime without the need for a full-fledged Just-In-Time compiler. The proposed approach allows us to apply profitable optimizations that could not be used at compile-time due to lack of sufficient information. It also enables us to optimize code fragments multiple times, depending on the specific conditions of the execution environment at runtime, while limiting the performance overhead to a negligible amount.

6.1 Introduction

When dealing with parallel languages, compilers have been used in a classical way: they perform translation to machine code, that is later executed. Leveraging on the structure of the input language, some compilers perform aggressive optimizations, such as work-items pre-scheduling [85, 135]. The generated code is always oblivious to the existence of a compiler, and the compiler, even when it is a JIT, does not provide any kind of "service" to the program.

This work presents a more dynamic approach to compilation. We generate code meant to interact with the compiler during execution, to exploit dynamically available information to optimize the code on-the-fly, without the burden of a full-fledged JIT system. In our approach the compiler works together with a micro-thread scheduler. *Pipeline stalls* in micro-threads containing the program code can be used to execute the compiler optimizers, thus *minimizing compiler overhead* at runtime.

The main motivation for pursuing this approach is code optimization: by running the compiler during code execution, more information is available, enabling more precise optimizations.

As a side effect, our approach would benefit software deployed in binary-only form, meant to run on many different hardware configurations (*e.g.:* binary packages used by Linux distributions). Allowing the program to customize itself without being compiled from a bytecode form at deploy-time and without the need for a Just-In-Time compiler

(that is time- and resource-consuming at runtime) could be useful in a variety of scenarios.

Section 6.2 of this chapter deals with existing approaches to runtime compilation and code modification. Section 6.3 introduces our approach to perform runtime compilation using lightweight compilation micro-threads and runtime scheduling. Section 6.4 discusses scenarios where our technique can be useful and Section 6.5 concludes.

6.2 Related Work

The problem of adapting programs to the runtime environment and to the specific set of data they are working on has been tackled in many ways, mostly related to the concept of dynamic compilation, also known as Just-In-Time (JIT) [21]. According to this approach, parts of a program are compiled while the program itself is being run, when more information is available than at compile time and can be used to perform further optimizations.

One of the first works on JIT systems [69] deals with the fundamental questions of JIT: determining what code should be optimized, when, and which optimizations should be used. We deal with similar questions, but we decide at compile time what code to optimize and which optimizations to apply, and postpone to runtime the task of answering "when" and "how" to optimize the code.

JIT compilation introduces an overhead in execution time because it causes the program to be idle while waiting for the new machine code. Considering that most programs spend the majority of time executing a minority of code [90], two papers [45, 47] independently proposed the approach called *mixed code*, where most of the code is interpreted and only the frequently executed part is identified, compiled and optimized at runtime.

Some works [78, 92] exploit multi-core processors to hide compilation latency: the compiler is run in a different thread and uses heuristics to predict the next method to compile before it is actually needed by the program. State of the art implementation can be found in [91].

All JIT-related works assume a compiler is running alongside the program. This causes a big overhead because of the memory and the time it takes to compile a new, optimized version of the code. On the other hand, the approach we propose does not need a full-fledged compiler running alongside the program. It only applies lightweight transformations to code specifically generated at compile-time to allow it, thus requiring much less resources, while being only slightly less flexible than

a full-fledged JIT compiler.

Staged compilation is another low-overhead approach. It splits the compilation in a static and a dynamic stage. The static one compiles "templates", building blocks for the dynamic stage that connects them and fills the holes left by the static stage with run-time values [104].

An example of a state of the art JIT compiler is the HotSpot Java Virtual Machine [91, 116], that uses adaptive optimization on top of a mixed code approach, with continuous monitoring and profiling of the program during its execution. It performs non-conservative optimizations, such as inlining frequently called virtual methods. To deal with Java's dynamic class loading it uses *dynamic de-optimization*. When the assumptions that led to a non-conservative optimization become false, the code is de-optimized back to a safe version, and then new optimizations are applied.

HotSpot supports two different compilers, namely the "Client" one and the "Server" one. The HotSpot Client Compiler [91] is focused on optimizing client applications, where the responsivity of the application is more important than deep optimization. The HotSpot Server Compiler [116] aims at optimizing the server applications, where it is worth spending more time compiling parts of the application. Therefore, the compiler features all the classic optimizations, as well as some Java specific ones.

A different approach to adapting programs to the runtime environment is *self-modifying code*. Von Neumann architectures [146] represent code in the same way as data. Therefore, a program is able to modify its own code while running, changing its own behavior. The main drawback of self-modifying code is the difficulty for many programmers to understand, write and maintain such code. Self-modifying code is used in [102] to write an operating system kernel and in Knuth's MIX architecture [89] for the subroutine calling convention.

6.3 Proposed Approach

We present a new kind of compiler optimizations, able to adapt to highly dynamic execution environments without adding excessive overhead at runtime.

Optimizations built according to our approach are divided in two phases, one to be executed at compile time and one at runtime. The runtime phase is extremely lightweight and is assigned the task of modifying the program to actually apply the optimization according to the current state of the execution environment, whereas the compile-time

phase has to generate the machine code of the program in such a way to allow this to happen

The need for offloading most of the optimization-related computation on the static compiler has already been assessed by other works, such as [112]. Another example of cooperation between compiler and runtime can be found in [67] for GPGPUs.

With respect to the traditional static/dynamic compilation flow, where compilation and execution phases are clearly separated, we have to face two specific issues.

Expected profitability Not all optimizations have to be delayed at runtime. We aim at applying an optimization at runtime only if *there are no sufficient information* to apply it at compile-time and a *considerable improvement* is expected. At the same time, since code is generated at compile-time, we free the runtime environment from the burden of applying trivial but needed optimizations, such as copy propagation, that a traditional JIT approach has to perform during program execution.

Moreover, delaying at runtime all applicable optimizations is not feasible, because we aim at keeping a lower overhead with respect to traditional JITs. This naturally leads to a careful selection of which optimizations to delay, based on their expected profitability.

Compiler interference Runtime application of optimizations leads to possible conflicts between optimizers and the running optimized code. This happens because there is no strong separation between the compiling and running phases of the program. To guarantee consistency, it is necessary to coordinate optimization and execution of the code.

To handle these issues, we define a model that allows us to detect, handle and apply profitable optimizations. We represent the program using a set of micro-threads (similar to those described in [51, 56, 87]). Part of these micro-threads are defined by the programmer or by the compiler and contain the code of the program being written. We call them *computational micro-threads*. The remaining micro-threads are called *compiler micro-threads*. They are generated by the compiler and contain the code that is able to apply optimizations at runtime.

Each compiler micro-thread is associated to a computational one, and manipulates one of its *optimizable region*s, that are the sections of the code of a computational micro-thread that can be modified by a runtime optimization. The dual of an optimizable region is an *optimizing region*. It is defined as all the code of a computational thread that is not part of the corresponding optimizable region. The optimizing region is the

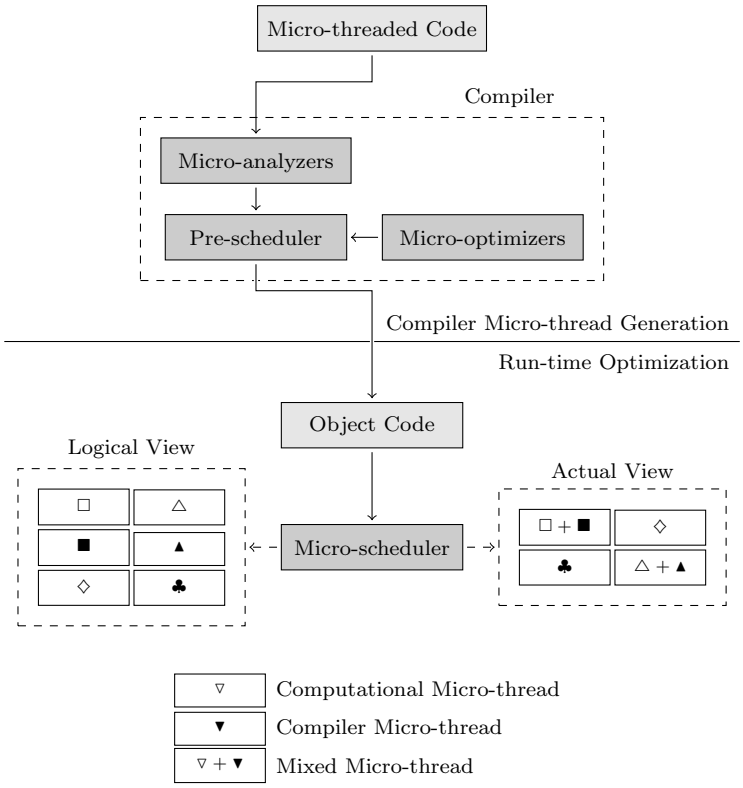


Figure 6.1: Proposed compilation/execution pipeline. Micro-threaded code is analyzed to detect profitable runtime optimizations. Compiler micro-threads (filled-in symbols) are built and possibly merged with computational micro-threads (empty symbols), generating mixed micro-threads (both symbols)

region where the optimizer micro-thread can safely run concurrently with the computational micro-thread to apply its optimization.

6.3.1 Compilation/Execution Pipeline

With reference to Figure 6.1, our compilation approach is split into two parts: *generation of compiler micro-threads* and *runtime optimization*.

The first step is intended to be part of a static compilation pipeline, and its goal is preparing the code to be optimized at runtime. We want to consider only optimizations that cannot be applied at compile-time, so this step should be run after standard compiler optimizations. First of all, it has to look at the input code to find candidate applicable runtime optimizations. It is not possible to apply all optimizations, because

interferences between them are possible. Therefore, they must be scored with respect to the expected profitability. Then, the model based on optimizable/optimizing regions allows us to represent such interferences on the computational micro-thread control flow graph. A pre-scheduler analyzes the interferences and selects the best optimizations. Finally, the corresponding compiler micro-threads are generated from a library of *micro-optimizers*. For each computational micro-thread, multiple compiler micro-threads can be generated, one for each optimization.

It is worth noting that the micro-threaded model is a purely logical one: we aim at minimizing runtime overhead, therefore if the system is implemented on a computing architecture with high costs of inter-thread communication the micro-threads can be multiplexed into a single mixed micro-thread. To do this, the pre-scheduler analyzes the computational micro-thread and compiler micro-threads, and schedules in a single mixed micro-thread the instructions from both, according to constraints imposed by optimizable and optimizing regions. Merging different micro-threads together was proven to be effective for scheduling Single Instructions Multiple Threads programs [95, 96, 135]. In our approach, micro-threads are not homogeneous, but we think that similar techniques have to be used to limit as much as possible the overhead of runtime optimizations.

The output of the pre-scheduler is a set of threads containing micro-threaded code intended to be run by a runtime micro-scheduler. From the logical point of view, the runtime scheduler has to manage both computational and compiler micro-threads, but, due to pre-scheduling, it actually has to manage mixed micro-threads too: therefore, some of the micro-threads need synchronization, whereas some other micro-threads have already been merged by the pre-scheduler, thus eliminating the need for explicit synchronization.

6.3.2 Run-time Optimization

The compiler micro-threads have to change the code of their associated computational micro-thread to optimize it. This can be done explicitly, using *self-modifying code*, or implicitly, using *branch tables*.

The compiler micro-thread is generated together with the optimizable region code it is associated to. Indeed, knowing the layout of the optimizable region, it is possible to generate instructions performing binary rewriting at runtime, without influencing other regions of code of the computational micro-thread.

The strength of self-modifying code is the ability to generate the most suitable instructions for a given optimizable region. However, the cost of

code morphing is considerable. An entire region of code must be rewritten. This requires editing the memory locations that store the optimizable region. Moreover, if the code is shared by multiple micro-threads, code cannot always be modified: the conditions triggering runtime optimization for a given micro-thread could be not valid for the others. Despite these limitations, self-modifying code can be an effective optimization strategy, if exploited for highly profitable optimizations, like inner loop vectorization.

A branch table, on the other hand, is a collection of unconditional jumps to different locations. At runtime, an index is used to select where to jump to. It can be implemented using different techniques, and is used to translate switch statements or to implement virtual tables. In our context, branch tables enable compiler micro-threads to change the execution flow of the associated computational micro-thread without changing its code. When our logical model is implemented, an optimizable region should be represented as a collection of sub-regions linked using branch table-based jumps. Compiler micro-threads just have to modify the indices used to select the active jump in branch tables, thus implicitly modifying the control flow graph of the computational micro-thread.

With respect to self-modifying code, branch tables impose less runtime overhead, since applying an optimization simply amounts to setting a set of indices. On the other hand, all the possible fragments of code used to optimize the region need to be generated at compile-time. The low runtime overhead makes this strategy suitable for highly dynamic scenarios, where the compiler micro-thread has to modify the execution flow more often.

To trigger an optimization, compiler micro-threads must observe the state of the associated computational micro-thread. If an optimization was postponed at runtime because the value of a variable was unknown at compile-time, the observed state will surely include that variable as one of the elements to be considered to decide when and how to apply the optimization at runtime.

It is worth noting that our approach enables a wide range of runtime optimizations. We use branch tables to restructure the execution flow and, where this is not sufficient, we also allow code morphing to apply deeper modifications. The use of branch tables should not be perceived as just a static branch prediction [23], since it is not performed statically, but is dynamically changed every time it is needed, as a result of modifications in the execution environment.

The strong relationship between computational and compiler micro-threads motivate us to emphasize the importance of having an effec-

tive and efficient pre-scheduler. Data related to a computational micro-thread must be collected and analyzed by the corresponding compiler micro-threads. Moreover, compiler micro-threads change the behaviour of the computational micro-thread. By scheduling the different micro-threads together we aim at avoiding communication delays between them. This guarantees deterministic interactions between micro-threads, as well as high performance. Even if it is strongly discouraged, our proposal does not prevent scheduling compiler micro-threads independently. However, in this case it is required to consider explicit synchronization between micro-threads, possibly exploiting weak memory consistency models [12] to limit communication overhead.

The authors of [99] observe that current production-quality compilers have issues with vectorization because the required analyses, such as interprocedural alias analysis, are not available. Such an analysis is really hard to implement at compile time, but pointers can be disambiguated at runtime. This further supports the need for splitting the compilation effort between compile-time and runtime, as allowed by our approach.

6.4 Foreseen Applications

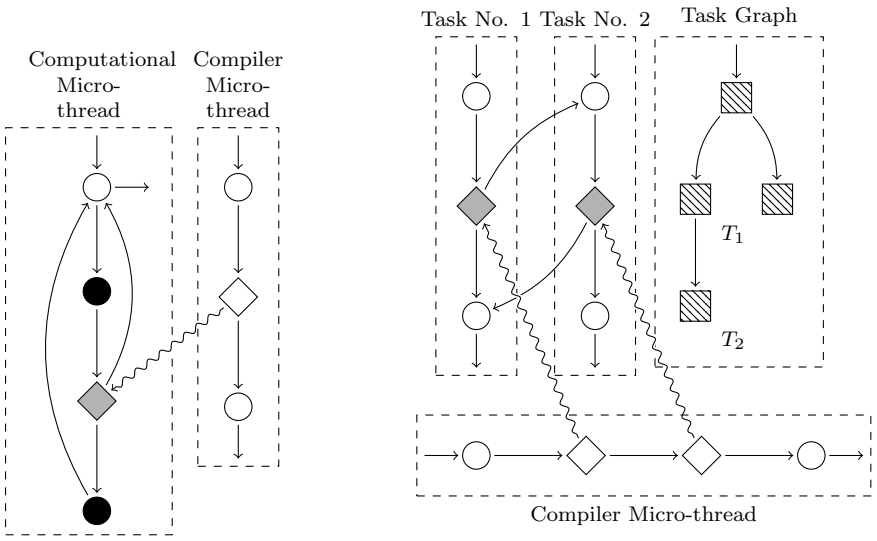
In this section we present two examples of optimization that would benefit from our approach. Figure 6.2 gives a brief overview.

6.4.1 Adaptive Loop Unrolling

The classic loop unrolling optimizations [22] can lead to improved, unaffected or worsened execution times depending on whether the right unroll factor is chosen [37, 46]. This is a challenging task, requiring good knowledge of the target architecture [129]. In most cases, this is only available at runtime, and is exploited using a JIT compiler. Unfortunately, JITs are really heavyweight and time consuming.

With our approach, we estimate a maximum sensible unrolling factor k at compile-time. We unroll the code of the loop k times and insert a branch table read between each pair of unrolled loop bodies, as in Figure 6.2(a). This is the optimizable region. At runtime, the compiler micro-thread determines the best unrolling factor $n \leq k$ (according to the size of caches, the number of required iterations, etc.) and modifies the n-th branch table read so that it jumps back to the loop header, and all the other ones so that they either jump to the next instruction, or are substituted by `nop` instructions.

This approach is much lighter that a full-fledged JIT, but it does not enable the application of further optimizations on the unrolled code.



(a) Adaptive loop unrolling

(b) Dynamic task fusion

Figure 6.2: Graphical representation of two foreseen applications of our proposed approach. White diamond nodes represents optimization actions, while gray diamond nodes are the optimized region

However, if the underlying architecture is micro-programmed, the machine code will be rewritten and optimized by the hardware, making our code comparable to that unrolled by a JIT.

6.4.2 Dynamic Task Fusion

Task based data-flow programming models have been proven to be an attractive way to tackle some parallel applications [133]: tasks are generated on the fly, thus they require the use of a runtime scheduler to select and start them according to data and control dependencies. Therefore, after each task finishes executing, control has to return to the scheduler so that it can start the next task.

Using our approach, we can define an optimizable region just before the end of the machine code of each task, made of just a branch table read. As shown in Figure 6.2(b), at runtime, a compiler micro-thread supports the scheduler: it observes the state of the system and modifies the corresponding branch table to have it point to the beginning of the code of the next ready task. Therefore, tasks can be executed continuously, without the overhead of reaching back to the scheduler at the end of each of them. The modification of the branch table takes place

as soon as the compiler micro-thread is aware of the next ready task, therefore the current and the next task will be executed one immediately after the other, as if fused together. Some call to the scheduler will still be needed, for example in order to mark a task as finished, unlocking the depending ones.

When the task graph is known at compile time, more aggressive optimizations can be performed [63]. Our approach does not allow this, but it limits the scheduling overhead that arises when inter-dependent tasks have to be executed (as tackled in [144]) and handles highly dynamic applications where the task graph is known only at run-time, even if the code is generated at compile-time.

6.5 Concluding Remarks

In this chapter we presented a novel lightweight approach to dynamically optimize parallel programs, based on the use of compiler micro-threads that modify the running program at runtime, adapting it to the current environment.

We described some optimizations that could implemented using our methodology, to show that is general enough to be applied to a wide variety of algorithms. At the same time, though, it does not need to be completely general-purpose, since it is not meant to completely replace other techniques, such as static optimization or JIT compilation.

7 Fault Sensitivity Analysis of Synchronization Primitives

Modern multi-core processors provide primitives to allow parallel programs to atomically perform selected operations. Unfortunately, the increasing number of gates in such processors leads to a higher probability of faults happening during the computation. In this chapter, we perform a comparison between the robustness of such primitives with respect to faults, operating at a functional level. We focus on locks, the most widespread mechanism, and on transactional memories, one of the most promising alternatives. The results come from an extensive experimental campaign based upon simulation of the considered systems. We show that locks prove to be a more robust synchronization primitive, because their vulnerable section is smaller. On the other hand, transactional memory is more likely to yield an observable wrong behaviour in the case of a fault, and this could be used to detect and correct the error. We also show that implementing locks on top of transactional memory increases its robustness, but without getting on par with that offered by locks.

7.1 Introduction

The dramatic increase of multi/many-core systems' complexity leads to extensive introduction of a wide variety of parallel applications and algorithms, and therefore to the necessity of efficient and safe ways to allow synchronization among threads. *Locks* are (historically) the first and still the most popular software synchronization primitive [75]. Using locks requires the programmer to carefully avoid several common mistakes in the design of massively parallel programs, that would lead to erroneous behavior such as starvation or deadlocks; moreover, it has been argued that lock-based synchronization would make actions such as modifications or extensions of existing programs more exposed to programming errors. To avoid these risks, Herlihy and Moss [74] proposed the *Transactional Memory* concept, allowing "lock-less synchronization". Nowadays many different implementations of transactional memory have been proposed, either software [76, 130] or hardware [15, 38, 68, 105] based.

Much effort has gone into discussing various developments for locks and transactional memory, focusing on performances and - more recently - on power consumption [59, 106]. Yet, a further aspect should be taken into account as well, namely, the impact of hardware faults on the synchronization solution. The increasing integration level, density and scaling in transistor dimensions will have great impact on the reliability of next-generation systems. A non-negligible amount of *"hard"* (permanent) faults is likely to affect the chip even during its working lifetime; decreasing geometries moreover make it more probable that *"soft"* (temporary) faults will affect some of the gates during computation. In particular, "upsets" affecting memory elements should be taken care of: given a set of processors concurrently executing a task, if one of the processors hangs because of a memory fault this could block all other processors waiting for synchronization with it, so that the erroneous behavior would propagate in a dramatic fashion throughout the system.

In this chapter, we aim at analyzing the effect of faults on synchronization primitives. In particular, we compare the sensitivity to faults of locking techniques and hardware transactional memory, adopting technology-independent fault assumptions for both cases and exploring - through extensive simulations - the outcome of comparable fault distributions. Many papers present fault analysis and handling for single threaded systems [123, 148], but to the best of our knowledge, this is the first investigation about the differences between these primitives with respect to fault sensitivity performed by using experimental results.

The rest of this chapter is organized as follows: Section 7.2 describes the faults we are considering, Section 7.3 presents the methodology we used for performing the simulations, Section 7.4 shows the experimental results and Section 7.5 concludes.

7.2 Faults characterization

Our focus is here on soft (transient) faults, more specifically on faults identified as "Single Event Upsets" (SEUs) [43]. For our purposes, we organize faults into two different classes: the ones that affect the *general computation* - here defined *general faults* - and the ones that specifically affect mechanisms related to *critical sections*, either protected by locks or by transactions. Critical sections are particularly sensitive sections of code that are present in multi-threaded programs: wrong access to one of them by one of the concurrent threads can produce relevant errors in the program and can cause deadlocks or starvations, leading to the

inability for the program to finish its execution. Hereafter we will only focus on critical section-related Single Event Upsets (SEU): we ignore other (general) types of faults, such as program counter corruption, that are beyond the scope of our analysis. Moreover, we aim at a technology-independent analysis: no assumptions are made here concerning the causes of faults, but we actually consider functional errors - affecting the outcome of specific instructions or operations. This will lead us to examine errors as affecting memory words or registers, often collapsing a number of different faults into one "equivalent" error type. In the same spirit, uniform random distributions will be adopted (thus abstracting from other possible distributions due to technological peculiarities).

According to Gawkowski et al. [60], the following outcomes can derive from applying faults to a program:

Correct Result (CR) The program correctly terminates its execution, computing the right value.

Incorrect Result (IR) The program gracefully terminates its execution, but the computed value is not correct and the system does not detect the error.

Fault Detected by the System (FDS) An hardware exception occurs. The system terminates the faulted program following predetermined policies.

Timeout (T) The program does not respect its timing requirements and is terminated by the system.

User-defined Message (UM) The program detects a misbehaviour, that is signalled to the user.

We follow the same classification, with the exception of User-defined Messages, since we did not add any error correction/detection machinery to the analyzed programs.

7.3 The Methodology Adopted

In order to obtain an indication of the respective performances of lock-based and transactional-memory-based solutions (as far as sensitivity to faults is concerned) we chose to set up an experimental environment (based on simulation tools) capable of simulating the operation of a

realistic multiprocessor system as well as of supporting fault injection and simulating behavior after fault.

This choice is due to the fact that the only viable alternative would be to perform an analysis starting from the netlist of a hardware device. This device should support both lock based and transactional synchronization primitives. Moreover, it should be a neutral, publicly available benchmark (a personal choice would risk to be biased). Since such a device was not available, we decided to go for a simulation approach, so as to provide at least a first analysis that, although less precise, is more general and a good starting point for further work.

To obtain the experimental results presented here, we started from the SESC simulator. SESC is "a microprocessor architectural simulator [...] that models [...] chip multiprocessors, [...]". CPUs used as nodes are MIPS processors, with "a full out-of order pipeline with branch prediction, caches, buses, and every other component of a modern processor necessary for accurate simulation" [124]. More specifically, SESC operates at at functional-block level simulating the execution of a program.

In order to support the simulation of parallel programs, SESC provides its own implementation of a POSIX-like threading library, called *libapp*; libapp is much simpler than pthread, but it provides all that is needed for the aim of the present work - at least insofar as lock-based synchronization is concerned. Namely, libapp provides fork/wait primitives and lock/unlock primitives. While this allows us to proceed with the analysis of fault impact on lock-based solutions, to perform our comparison we also need an implementation of a transactional memory - which is not provided by SESC.

On the other hand, SuperTrans [119], developed by University of Florida Advanced Computing and Information Systems Laboratory, is "a multicore, cycle-accurate and multiple issue simulator built on top of the SuperESCalar (SESC) framework that is capable of simulating three of the most common dimensions of hardware transactional memory (Eager/Eager [15, 105], Eager/Lazy [15, 122], Lazy/Lazy [68])". SuperTrans, just as SESC, is released as open source. It includes all that is part of SESC (therefore, the lock based management of memory) plus a transactional memory module. For these reasons, we chose SuperTrans as the tool for transactional-memory related simulations; being based on SESC, it granted the kind of consistency that was mandatory to compare results of simulations obtained on the two systems.

In order to explore the effects of faults, we modified SuperTrans by adding a new software component, that we named *fault injector*, allowing us to specify where and when to inject faults during the simulation, so that we can observe the outcome of the management of the mutual

exclusion between two or more processes trying to access a single critical section. The fault injector can support an arbitrary number of faults. The characteristics of the faults can be completely specified by the user or randomly generated.

7.4 Impact of Faults on Synchronization Mechanisms

In order to evaluate how faults affect the behavior of programs run by systems that use, respectively, locks or transactional memory to protect the critical sections, we carried out an extensive experimental campaign, using a small set of synthetic benchmarks (depicted in Table 7.1) that implement well known concurrency problems, such as *shared counter* or *reader/writer interactions* [136]. Using such simple examples allows us to easily inject faults exactly in the registers and cache lines that will be accessed by the code while inside a critical section. Therefore we can verify the effect of faults on the more likely sources of problems related specifically to the synchronization mechanism adopted rather than to the general effects of faults on program's execution. Moreover, these small benchmarks share the same structure of most complex concurrent applications, so that the results we obtain are actually general. Studying the effect of faults on synchronization primitives has a direct impact on determining how the behaviour of the application will change because of them. In fact, many years of research on operating systems [136] prove the importance of the correct behaviour of such primitives.

We will now describe in detail how faults are injected in the micro-benchmarks and what are the results obtained using the *SC* micro-benchmark as a running example. Section 7.4.1 reports on fault injection in lock-based critical sections, while Section 7.4.2 refers to transactional-memory-based critical sections, Section 7.4.3 describes faulting critical sections protected with transactional-memory-based locks - a solution that, while non-realistic, allows completing our fault-related analysis with this alternative derived from the two basic criteria. Finally, Section 7.4.4 presents the experimental campaign setup and its results.

7.4.1 Lock-based Critical Sections

From the users perspective, protecting a critical section *cs* requires invoking a LockAcquire function before entering *cs*. This guarantees that no more than one thread at a time enters the critical section. To leave *cs*, a thread must invoke a LockRelease function. This allows

Algorithm: ShrdIncLock
Input: a shared counter *cnt*
Result: *cnt* safely incremented by 1

1 LockAcquire (*cnt.lock*)
2 $cnt.n \leftarrow cnt.n + 1$
3 LockRelease (cnt.lock)

Figure 7.1: Shared counter update. Locking functions guarantee *mutual exclusion* between threads while concurrently incrementing the counter

Algorithm: ShrdIncTrans
Input: a shared counter *cnt*
Result: *cnt* safely incremented by 1

1 TransBegin ()
2 $cnt \leftarrow cnt + 1$
3 TransCommit ()

Figure 7.2: Shared counter update exploiting transactional memory. If a conflict is detected during a transaction, it is aborted and restarted by the hardware

other threads to access *cs*. Figure 7.1 shows how these routines can be employed to safely increment a shared-counter.

Such locking/unlocking routines are built on top of hardware synchronization instructions, such as *atomic eXCHanGe*, *Compare And Swap*, and *Load Linked/Store Conditional*. No other *ad hoc* hardware capabilities are exploited to implement the routines: the remaining code segments are implemented using standard instructions. Figures 7.3 and 7.4 show LockAcquire and LockRelease routines respectively.

Any fault generated inside a critical section can corrupt the current thread's private data, as well as the private data of other threads and shared data. This happens because a critical section's body contains only non lock-related instructions and the locking algorithm has no knowledge of the data accessed and of the instructions executed inside it.

If we consider critical section boundaries, identified by LockAcquire and LockRelease routines, we see that a fault affecting data accessed by these routines is catastrophic because they control the *access to the critical section*. Even in the presence of transient faults, the program behavior is radically modified: more than one thread will access the critical section at the same time, performing a computation at the *wrong time*. The faulted program behavior matches classical concurrent programming errors, such as *lost update*, *dirty read/write*, ...

For our experimental campaign, we start by injecting faults affecting LockAcquire. The most important operation performed here is Xchg (Figure 7.3, Line 2): it atomically replaces the memory word where *lock* resides with the *LCK* constant, returning the value stored there

Algorithm: LockAcquire
Input: a lock *lock*
Result: *lock* locked by current thread

```
1 val ← Xchg (lock, LCK)
2 while val = LCK do
3   repeat Nop until lock ≠ LCK
4   val ← Xchg (lock, LCK)
```

Figure 7.3: Implementation of LockAcquire

Algorithm: LockRelease
Input: a lock *lock*
Result: *lock* unlocked

```
1 lock ← UNLCK
```

Figure 7.4: Implementation of LockRelease

before the swap took place. We identify three elements such that faults affecting them are critical for the synchronization process, namely: *lock*, the register containing the LCK constant, and the return value.

To emulate faults on *lock* we consider them just by their outcome: having the program reading/writing the wrong memory location, therefore causing Xchg to return a wrong value. Such value is later read (Figure 7.3, Line 2) by a comparison instruction to detect whether to enter the critical section, so this fault can allow the current thread to enter the critical section, even if the lock is not held. The program behavior cannot be predicted, and both CR and IR can be observed. A write on the wrong address could be detected, depending on the specific address, if a FDS situation (e.g. segmentation fault) occurs.

Altering the LCK word results in writing the wrong marker in the *lock* memory location. If it turns out to be equal to the $UNLCK$ marker, the current thread enters the critical section without the other threads being aware that the lock has been taken. Therefore, they can enter the section too, leading to wrong behaviour. We can observe the same behaviour also if the written marker is invalid, because every value not equal to LCK allows entering the critical section. We can observe CR if the dynamic schedule does not result in a data race, IR or FDS otherwise.

A transient fault on the return value can result in two different behaviours: if the faulted return value is equal to LCK, the current thread spends some cycles (Figure 7.3, Lines 2 and 3) waiting for the lock to be released, without corrupting data. Otherwise, the current thread enters the critical section, incurring into a potential data race. We can observe the same program behaviour as in the previous case: CR, IR, or FDS.

As a final remark, it is worth noting that in the case of a thread trying to enter a critical section it is very unlikely to incur into T behaviour

(provided only transient faults are applied, as in our experiments). For this to happen, the value accessed through the *lock* variable (Figure 7.3, Line 2) should always be equal to the value of the LCK constant: this requires either to continuously fault *lock* in such a way to end up reading from memory locations containing the LCK value, or to fault the return value of the XCHG instruction every time in such a way that it results equal to LCK. Similar considerations apply to the spin wait loop, too (Line 3).

The LOCKRELEASE routine is a simple store to memory. Its behaviour can be altered by injecting faults on *lock* and on the $UNLCK$ marker. Modifying *lock* shows the same behaviour as writing to an invalid memory address, potentially generating CR, IR, and FDS behaviours.

Finally, a fault affecting $UNLCK$ results in generating an invalid marker that corrupts *lock*, but the locking algorithm is not influenced: the first thread entering into the critical section restores *lock* to a consistent state. On the other hand, writing the valid but incorrect value LCK results on T behaviour: the lock is released incorrectly, preventing any thread from entering the critical section.

7.4.2 Transactional Memory-based Critical Sections

In order to protect a critical section using transactional memory, the user employs three routines: TRANSBEGIN (instructing the transactional memory to save the current context), TRANSCOMMIT (to publish the memory operations performed), and TRANSABORT (to explicitly terminate and restart a transaction). Figure 7.2 shows how transactional memory can be used to protect a shared counter update.

In transactional memory approach, critical section access control is distributed; every memory operation inside a critical section is validated by the transactional controller in order to detect conflicts. Detection is performed by analyzing the *read set*, (the set of memory locations read by a thread), and the *write set*, (the set of memory locations written by a thread). To emulate errors corrupting the *read set* as well as the *write set*, requires one can collapse the various fault causes into faults affecting the addresses manipulated by the transactional controller. Therefore, we will inject faults near memory access opcodes so as to affect the system immediately before memory access.

Corrupting the read set can be modeled as reading from the wrong memory location. A transactional load, LWX, both interacts with the transactional controller and fetches data from memory. As a result, the read set of the faulted processor becomes inconsistent and a wrong value is read from the memory. If the wrong value is used for subsequent

computations, it can produce IRs. The same behaviour can occur even if the read value does not directly produce a corrupted value. In fact, the transactional controller could be unable to detect a conflict due to the corrupted read set, thus allowing a transaction to commit when it should have been aborted instead. Moreover, if the corrupted address is later used for a memory store to a location not accessible by the faulted processors, a FDS occurs.

If faults lead to corrupting the write set, the same behaviour can observed. In this case the faulted instruction is the transactional write, SWX; as in the case of LWX, the instruction also interacts both with the transactional controller and the memory. Depending on the fault-affected value, an IR or FDS can occur. The difference with respect to faulting the read set is that an FDS can occur immediately.

Hardware implementations of transactional memory introduce three new opcodes, namely XBEGIN, XCOMMIT, and XABORT respectively implementing TRANSBEGIN, TRANSCOMMIT, and TRANSABORT. All these operations do not use general purpose hardware; they interact directly with the transactional memory controller, thus to simulate faults relative to them we cannot just inject faults into registers or into non-transactional memory, but we have to fault the simulated hardware primitives themselves. Faults concerning this scenario corrupt processor context saved by XBEGIN and restored by XCOMMIT and XABORT. Since these faults would be very much dependent on a specific implementation and technology, we do not consider them; obviously, extending the set of faults would increase the sensitivity to faults of the system.

7.4.3 Transactional Locking-based Critical Sections

As shown in Section 7.4.1, we can inject a wide variety of faults on locks, but the lock is directly manipulated only at critical section bounds, so there is not much possibility for such faults to happen. Every other fault happening inside a critical section protected by locks is not related to locks themselves: as such it could happen whatever the synchronization primitive being used, and is therefore not interesting for this study. On the other hand, transactional memory is vulnerable to a narrow class of faults, see Section 7.4.2, but they expose more faulting opportunity because as long as the transaction is active every memory access could be influenced by faults in the *read set* or the *write set*.

This observation led us to try to implement locks "on top of" transactional constructs. While this is not a viable solution for real systems, it allows us to study whether transactional memory helps reducing faulting opportunities. The locking and unlocking algorithms are the same

Algorithm: TransXchg
Input: an address *addr*
a value *val*
Result: *val* written into *addr*,
old value returned

```
1 TransBegin ()
2 old ← mem[addr]
3 mem[addr] ← val
4 TransCommit ()
5 return old
```

Figure 7.5: Atomic exchange implemented using transactional memory. It is used as a building block for transactional memory-based locks

Algorithm: TransRelease
Input: a lock *lock*
Result: *lock* unlocked

```
1 TransBegin ()
2 lock ← UNLCK
3 TransCommit ()
```

Figure 7.6: Lock release routine implemented on top of transactional memory. A normal store cannot be used because all lock-related operations must be put under control of the transactional memory controller

used for lock-based critical sections (Figure 7.3 and 7.4). In order to exploit transactional memory, we replaced the Xchg instruction with an equivalent routine written using transactional constructs. Its implementation can be seen in Figure 7.5. The lock release routine, reported in Figure 7.6, has been modified so as to be protected by a transaction.

For faults happening inside the critical section we can make the same observations as for locks, because the critical section does not contains any special instruction.

On the other hand, we note that injecting faults on critical section boundaries requires injecting faults on the transactions protecting the atomic exchange. The kind of faults that can be injected are the same as for transactional memory: basically, we can fault the read set and the write set.

In this particular critical section, the read set and the write set are identical: they consist just of the word used to store the lock. Faulting the lock address can thus produce FDS, IR, or T behaviours. The first arises when the faulted lock address refers to a memory region that cannot be written by the faulted thread. If the word identified by the faulted address can be written, a data race can occur, possibly generating either CR or IR. The T behaviour occurs when reading from the faulted address causes the faulted thread to spend too much time in the lock busy-wait loop.

Table 7.1: Fault sensitivity analysis benchmarks

SC	concurrent increment of a shared counter. Each thread performs 8 atomic increments.
SMC	concurrent increment of shared counters. Each thread executes 4 critical sections, incrementing 16 counters each time.
RW	reader/writer problem. Threads are partitioned into two equally sized sets: readers and writers. Writers produce items writing them into a global buffer. Readers read items from the buffer. When all items have been produced, the readers concurrently write all the read items into another buffer read by the main thread to perform a final sanity check. Buffers are implemented using arrays.
RWL	reader/writer problem. Same behaviour of *RW*, but shared buffers are implemented using single-linked lists.

7.4.4 Results of the Experimental Campaign

Our experimental campaign focused on the micro-benchmarks reported in Table 7.1. We coded each micro-benchmark in three different flavour, each employing a different primitive to protect its critical sections. The *lock* flavour, uses locks, *trans* uses transactions, while *trans-lock* adopts locks implemented by means of transactions.

Each micro-benchmark was first run without applying any faults. Observing the execution trace we detected points where faults could be injected, as suggested in Section 7.4.1, 7.4.2, and 7.4.3. Each flavour exposes different faultable points. Faults will affect execution with the *lock* flavour while acquiring and releasing the lock. The *trans* flavour is faultable while accessing the read set and the write set, i.e. near each LWX and SWX. The *trans-lock* flavour exposes the same faultable points as *trans*, but the critical section is shorter.

We aim at observing the evolution of the behaviour of the micro-benchmarks subject to an increasing number of faults. Therefore, for each of them we injected an increasing number of faults, from 1 to 4. For each benchmark, for each given number of faults, we performed

```
1 [upReg]
2 generator = 'uniform'
3 regNo = 'R18'
4 kind  = 'bitFlip'
5 atTime = 1100
```

Figure 7.7: An example of fault taken from the configuration file. A bit-flip fault named `upReg` will be applied to register `R18` at 1100^{th} cycle of the simulation

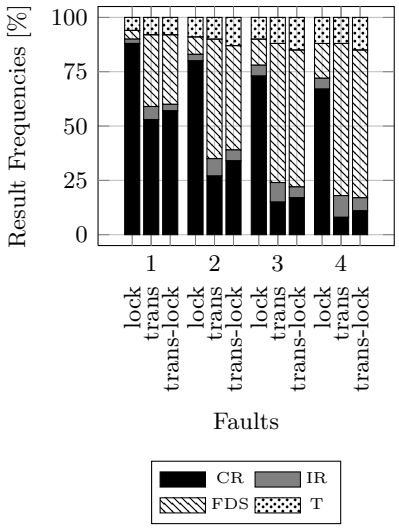


Figure 7.8: Distribution of benchmark results, varying the number of applied faults

960 [1] runs. Before each run we randomly extract i faultable points taken from those observed by analyzing the execution trace. To allow for some randomness, each fault was randomly applied between 1 and 4 cycles before the time instant it was registered in the original execution trace. In case of faults applied to registers, we randomly generated the number of the register bit to fault. For faults applied to cache line reads, we randomly generated the loaded word bit to fault. Figure 7.7 shows a generated fault entry in the SESC configuration file format.

Table 7.2 reports individual benchmark results, while Figure 7.8 shows the percentage of *CR*, *IR*, *FDS*, and *T* for each flavour, varying the number of applied faults.

The *lock* flavour is the most robust, because there are fewer points where a fault can be injected. Moreover, the fault must be injected at a precise time, otherwise the locking algorithm tends to mask the fault and thus overcomes a previous soft fault. In fact, the locking algorithm usually rewrites the content of the lock word at the beginning of the critical section, while trying to acquire the ownership via the XCHG instruction, and at its end, while releasing the lock. Moreover, not all faults injected on locks can be observed, because even if two threads happen to enter in a critical section at the same time, they could not

[1] 240 for RWL-trans.

Table 7.2: Results of fault injection on benchmarks

Benchmark		**CR** [%]				**IR** [%]			
		F_1	**F_2**	**F_3**	**F_4**	**F_1**	**F_2**	**F_3**	**F_4**
SC	lock	82	76	71	63	3	5	6	9
	trans	50	25	13	7	8	12	14	14
	trans-lock	55	35	18	10	5	6	8	8
SMC	lock	92	86	81	80	4	4	6	5
	trans	51	24	14	10	8	10	12	14
	trans-lock	57	34	21	13	6	10	10	11
RW	lock	87	77	68	61	3	3	4	5
	trans	59	33	18	9	3	4	4	5
	trans-lock	58	30	12	11	1	2	7	2
RWL	lock	89	83	73	64	0	1	2	3
	trans	48	19	11	4	3	2	3	3
	trans-lock	56	35	17	10	1	2	2	5
		FDS [%]				**T** [%]			
		F_1	**F_2**	**F_3**	**F_4**	**F_1**	**F_2**	**F_3**	**F_4**
SC	lock	4	9	13	17	11	10	10	11
	trans	42	63	73	79	0	0	0	0
	trans-lock	40	59	74	82	0	0	0	0
SMC	lock	3	8	9	11	1	2	4	4
	trans	40	64	73	74	1	2	1	2
	trans-lock	37	55	69	75	0	1	0	1
RW	lock	4	9	15	19	6	11	13	15
	trans	19	38	46	56	19	25	32	30
	trans-lock	25	38	54	59	16	30	27	28
RWL	lock	4	8	10	16	7	8	15	17
	trans	27	48	60	64	22	31	26	29
	trans-lock	26	39	54	58	17	24	27	27

incur in a data race, depending on the specific scheduling taking place.

Looking at Table 7.1 we see that the *trans* flavour obtains the worst outcome, with less CR compared to the *lock* flavour, because transactional memory exposes more faultable points. However, the probability that a fault will be detected (FDS) is greater, because most failures are due to accesses to wrong memory areas. These are detected by the operating system and could, in principle, be used to perform error correction, thus increasing the number of correct results.

Finally, implementing lock on top of transactional memory, i.e. the *trans-lock* flavour, increases the robustness with respect to *trans*, because each transaction lasts only as long as needed to change the lock value, but it cannot achieve the robustness of the *lock* flavour, because as short as that time span can be, every single access to memory during it can be subject to faults. Let us now analyse in detail the outcome of each benchmark.

Shared Counter and Shared Multi Counter The critical section associated to *SC* is the shorter of all the benchmark suite, while *SMC* employs a longer critical section, updating more than a shared counter at time.

The *lock* flavour is the most susceptible to short critical sections. Indeed, on such scenario the program hot spot is *lock acquisition*, so any fault that induces spending some extra cycles in the lock waiting loop, greatly lowers performance, generating a considerable amount of T behaviour. When increasing the length of the critical section, the number of T behavior decreases, as shown by the *SMC* micro-benchmark, where we can observe a greater number of CR.

Both *trans* and *trans-lock* flavours follow the same trends in both *SC* and *SMC*. To obtain T behaviour, read and/or write sets of a transaction must be faulted in order to read/write data from/to the shared data, forcing an abort. Both the micro-benchmarks have a small amount of shared data, so the probability of this outcome is negligible.

Reader/Writer and Reader/Writer List The *RW* micro-benchmark uses arrays to implement shared buffers, while *RWL* relies on single-linked list, thus critical sections are longer and access memory more frequently.

Locking-based techniques exhibit the same behaviour in the two micro-benchmarks. On the other hand, the *trans* flavour is heavily influenced by using single-linked lists. Using more complex structures results in more memory accesses, mostly related to list navigation. Thus, the

probability of incurring into a *FDS* increases.

7.5 Concluding Remarks

In this chapter we analyzed the behavior of locks and transactional memory when they are affected by faults. We injected from 1 up to 4 faults during the execution of selected benchmarks and analyzed the outcome of the execution. As it is easy to understand, while the number of faults grows, the probability of a visible failure increases. The important result is that locks proved to be more fault resilient because they expose a smaller "faultable surface", and it is therefore more unlikely for a fault to have the execution fail. We did not consider a specific hardware implementation, and focused only on observing the functionality of the synchronization primitives under the effect of faults.

Nonetheless, it should be considered that transactional memory requires specialized hardware components to be added to the system, and this components are themselves subject to faults. This suggests that the actual fault tolerance of transactional memory could be lower than our results suggest. Further experimental campaign should be conducted in order to prove this point. On the other hand, our results show that with transactional memory the system is more likely to be able to detect the presence of faults. Further experiments could determine whether fault detection and recovery capabilities could be more effective or easier to implement in a transactional memory based system.

8 Concluding Remarks

Current trend in computer architectures is to exploit the available die area to integrate more than one processing unit in the same integrated circuit. Thus, we are moving from architectures characterized by few powerful cores, to designs that employ multiple simpler cores. From the hardware perspective, overall performance is increased without requesting huge power budgets.

However, with respect to previous designs, programming parallel architectures is a challenging task. Indeed, problems due to incorrect coordination between available processing units constitute an hazard difficult to tackle. Moreover, achieving good performance requires perfect knowledge of the target architecture.

Parallel programming models deal with these problems by providing a simplified view of the parallel hardware. This allows to avoid most of the low-level issues due to incorrect synchronization and to achieve reasonable performance. However, this simplification often prevents full exploitation of the target architecture. This dissertation main goal was to investigate such problems.

In Chapter 3 an analysis of a widely used computational pattern involving barrier synchronization, *reduction computation*, has been performed in order to check for possible optimizations. We have shown that executing the barrier operation together with reduction computation allows to mitigate bottlenecks introduced by synchronization.

Later, we considered the problem of efficient data access. Indeed, modern parallel architectures employ NUMA memory subsystems, hence schedule of parallel computations to minimize the expected memory access penalty is mandatory for achieving good performance. This problem has been analyzed from the OpenMP perspective in Chapter 4 – to *schedule parallel loop* iterations – and from the MapReduce perspective in Chapter 5 – to *assign map tasks* to worker machines.

From a compiler perspective, aggressive optimization of parallel programs is difficult due to the lack of a reference parallel architecture. Indeed, performance of parallel code is highly dependent on the configuration of the underlying architecture. Thus, the compiler cannot setup an accurate cost model that can be used to evaluate the effectiveness of optimizations for a wide range of machines. Usually, using a JIT

compiler allows to address this issue. However, it has to be run just before the application is started, thus incurring in performance penalties. In Chapter 6 we have proposed a compilation scheme that splits the duty of program optimization between an analysis part performed at compile-time and an optimization part performed at run-time. The run-time optimization must be very *lightweight* and must enable applying optimization with low overhead.

Finally, due to the increased probability of soft faults in modern parallel architectures, in Chapter 7 we have conducted a study about how *faults* affect the execution of critical sections, protected by means of locks or by exploiting hardware transactional memory. The study is motivated by the fact that many programming errors related to parallel computing involve incorrect usage of synchronization primitives, hence the interest on understanding the impact of faulted synchronization primitives on the overall program.

In this dissertation, we have focused on improving the performance of basic building blocks found in every parallel programming model. Whilst such building blocks can help the development of lightweight runtimes, able to efficiently coordinate the access to data, a key question is represented by the expressiveness of such programming models. Indeed, the performance of hand-tuned MPI code is not matched by any available parallel programming model, due to simplifications introduced to ease programmers life.

From this dissertation point of view, little can be done to further improve such basic blocks. Indeed, most of parallel architectures are built up many simple sequential cores, which communication mechanisms are still based on fundamental synchronization primitives and uniform data access. While the engineering practice of replicating a functional unit in order to improve the performance of a system as a whole has been demostrated effective in many engineering field, in the context of computer architectures a leap is requested in order to move the computing model towards a more collaborative environment. Hardware transactional memory is an example of such move, however, it presents many drawbacks – e.g. high power consumption, scalability, fault sensitivity – mainly due to the fact that the execution model is still too much sequential.

An interesting future challenge is finding the right language abstractions that allow to match multiple goals: i) ease programmers from considering the hardware configuration of the target machine, such processors interconnet topology and memory hierarchy ii) enable aggressive optimizations by the compiler iii) solve the problem of performance portability, that currently represents a major drawback of parallel pro-

grams Our insight is that these goals can be reached only by enforcing the collaboration between programmer, compiler, and runtime. The next generation parallel programming model should be able to hide hardware details to programmers using an high-level interface allowing the application of aggressive optimization by the compiler. We also think that the compiler should be kept into consideration also at runtime, an objective that can be satisfied only by improving current JIT technologies to make their overhead acceptable.

Bibliography

[1] GNU libgomp, 2010. URL http://gcc.gnu.org/onlinedocs/libgomp/.

[2] AMD Fusion™ Family of APUs: Enabling a Superior, Immersive, PC Experience, 2012. URL http://www.amd.com/us/Documents/48423_fusion_whitepaper_WEB.pdf.

[3] Intel ARK Products Information, 2012. URL http://ark.intel.com/.

[4] Intel Many Integrated Core (MIC) Architecture – Advanced, 2012. URL http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[5] Intel™ Thread Building Blocks for Open Source, 2012. URL http://threadingbuildingblocks.org.

[6] NVIDIA CUDA™, 2012. URL http://www.nvidia.com/object/cuda_home_new.html.

[7] NVIDIA GeForce GTX 200 GPU Architectural Overview, 2012. URL http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf.

[8] GeForce 256 – The World's First GPU, 2012. URL http://www.nvidia.com/page/geforce256.html.

[9] NVIDIA GeForce 8800GT, 2012. URL http://www.nvidia.com/object/product_geforce_8800_gt_us.html.

[10] NVIDIA's Next Generation CUDA™ Compute Architecture: Kepler™ GK110, 2012. URL http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[11] OpenHMPP, new Standard for Many-core, 2012. URL http://www.openhmpp.org.

[12] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[13] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, second edition, 2009.

[14] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specifications, version 0.954*. Sun Microsystems, Inc., 2006.

[15] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *HPCA*, pages 316–327. IEEE Computer Society, 2005. ISBN 0-7695-2275-0.

[16] *OpenMP Application Program Interface, version 3.0*. ARB, 2008. URL http://www.openmp.org.

[17] *ARMv7 Architecture Reference Manual*. ARM, 2010.

[18] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, 2006.

[19] Vishal Aslot and Rudolf Eigenmann. Performance Characteristics of the SPEC OMP2001 Benchmarks. *SIGARCH Computer Architecture News*, 29(5):31–40, 2001.

[20] Vishal Aslot, Max J. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In Rudolf Eigenmann and Michael Voss, editors, *WOMPAT*, volume 2104 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2001. ISBN 3-540-42346-X.

[21] John Aycock. A Brief History of Just-In-Time. *ACM Comput. Surv.*, 35(2):97–113, 2003.

[22] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[23] Thomas Ball and James R. Larus. Branch Prediction For Free. In *PLDI*, pages 300–313, 1993.

[24] Pieter Bellens, Josep M. Pérez, Rosa M. Badia, and Jesús Labarta. Memory – CellSs: a Programming Model for the Cell BE Architecture. In *SC*, page 86. ACM Press, 2006. ISBN 0-7695-2700-0.

[25] Andrea Di Biagio, Ettore Speziale, and Giovanni Agosta. Exploiting Thread-Data Affinity in OpenMP with Data Access Patterns. In *Euro-Par (1)*, pages 230–241, 2011.

[26] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In Andreas Moshovos, David Tarditi, and Kunle Olukotun, editors, *PACT*, pages 72–81. ACM, 2008. ISBN 978-1-60558-282-5.

[27] John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. Extending OpenMP for NUMA Machines. In *SC*, 2000.

[28] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In Peri L. Tarr and William R. Cook, editors, *OOPSLA*, pages 169–190. ACM, 2006. ISBN 1-59593-348-4.

[29] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, 1999.

[30] Shekhar Borkar. Thousand Core Chips – A Technology Perspective. In *DAC*, pages 746–749, 2007.

[31] Edward Bortnikov. Open-source Grid Technologies for Web-scale Computing. *SIGACT News*, 40(2):87–93, 2009.

[32] Eugene D. Brooks. The Butterfly Barrier. *Int. J. Parallel Program.*, 15(4):295–307, 1986.

[33] François Broquedis, François Diakhaté, Samuel Thibault, Olivier Aumage, Raymond Namyst, and Pierre-André Wacrenier. Scheduling Dynamic OpenMP Applications over Multicore Architectures. In *IWOMP*, volume 5004 of *LNCS*, pages 170–180. Springer, 2008. ISBN 978-3-540-79560-5.

[34] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective. In *IWOMP*, volume 5568 of *LNCS*, pages 79–92. Springer, 2009. ISBN 978-3-642-02284-5.

[35] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Productive Cluster Programming with OmpSs. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par (1)*, volume 6852 of *Lecture Notes in Computer Science*, pages 555–566. Springer, 2011. ISBN 978-3-642-23399-9.

[36] J. Mark Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *In Proceedings of First European Workshop on OpenMP*, 1999.

[37] Steve Carr and Ken Kennedy. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. *ACM Trans. Program. Lang. Syst.*, 16(6):1768–1810, 1994.

[38] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *HPCA*, pages 97–108. IEEE Computer Society, 2007.

[39] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-parallel Pipelines. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 363–375. ACM, 2010. ISBN 978-1-4503-0019-3.

[40] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy H. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *MASCOTS*, pages 390–399. IEEE, 2011. ISBN 978-1-4577-0468-0.

[41] Huang Chun and Yang Xuejun. Improve OpenMP Performance by Extending BARRIER and REDUCTION Constructs. In Alexan-

der V. Veidenbaum, Kazuki Joe, Hideharu Amano, and Hideo Aiso, editors, *ISHPC*, volume 2858 of *Lecture Notes in Computer Science*, pages 529–539. Springer, 2003. ISBN 3-540-20359-1.

[42] Phillip Colella. Defining Software Requirements for Scientific Computing, 2004.

[43] Cristian Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4):14–19, 2003.

[44] *Chapel Language Specification, version 0.91.* Cray Inc., 2012.

[45] R. J. Dakin and Peter C. Poole. A Mixed Code Approach. *Comput. J.*, 16(3):219–222, 1973.

[46] Jack W. Davidson and Sanjay Jinturkar. Aggressive Loop Unrolling in a Retargetable Optimizing Compiler. In *CC*, pages 59–73, 1996.

[47] J. L. Dawson. Combining Interpretive Code with Machine Code. *Comput. J.*, 16(3):216–219, 1973.

[48] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[49] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. *Commun. ACM*, 53(1):72–77, 2010.

[50] Keith Diefendorff, Pradeep K. Dubey, Ron Hochsprung, and Hunter Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, 2000.

[51] Alejandro Duran, Roger Ferrer, Eduard Ayguadé, Rosa M. Badia, and Jesús Labarta. A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks. *International Journal of Parallel Programming*, 37(3):292–305, 2009.

[52] Speziale Ettore and Michele Tartara. A Lightweight Approach to Compiling and Scheduling Highly Dynamic Parallel Programs. In *HotPar'12 (Poster)*, 2012.

[53] Michael J. Fischer, Xueyuan Su, and Yitong Yin. Assigning Tasks for Efficiency in Hadoop: Extended Abstract. In *SPAA*, pages 30–39, 2010.

[54] Michael J. Flynn. Very High Speed Computing Systems. *Proceedings of the IEEE*, 54:1901–1909, 1966.

[55] Eric Freudenthal and Allan Gottlieb. Process Coordination with Fetch-and-Increment. In *ASPLOS*, pages 260–268, 1991.

[56] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In Jack W. Davidson, Keith D. Cooper, and A. Michael Berman, editors, *PLDI*, pages 212–223. ACM, 1998. ISBN 0-89791-987-4.

[57] Karl Fürlinger, Michael Gerndt, and Jack Dongarra. Scalability Analysis of the SPEC OpenMP Benchmarks on Large-scale Shared Memory Multiprocessors. In Yong Shi, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *International Conference on Computational Science (2)*, volume 4488 of *Lecture Notes in Computer Science*, pages 815–822. Springer, 2007. ISBN 978-3-540-72585-5.

[58] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, first edition, 2008.

[59] Epifanio Gaona-Ramírez, Rubén Titos-Gil, Juan Fernández, and Manuel E. Acacio. Characterizing Energy Consumption in Hardware Transactional Memory Systems. In *SBAC-PAD*, 2010.

[60] Piotr Gawkowski, Janusz Sosnowski, and B. Radko. Analyzing the Effectiveness of Fault Hardening Procedures. In *IOLTS*, pages 14–19. IEEE Computer Society, 2005. ISBN 0-7695-2406-0.

[61] Brice Goglin and Nathalie Furmento. Enabling High-performance Memory Migration for Multithreaded Applications on LINUX. In *IPDPS*, pages 1–9, 2009.

[62] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman P. Amarasinghe. A Stream Compiler for Communication-exposed Architectures. In Kourosh Gharachorloo, editor, *ASPLOS*, pages 291–303. ACM Press, 2002. ISBN 1-58113-574-2.

[63] Michael I. Gordon, William Thies, and Saman P. Amarasinghe. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In John Paul Shen and Margaret Martonosi, editors, *ASPLOS*, pages 151–162. ACM, 2006. ISBN 1-59593-451-0.

[64] Paolo Roberto Grassi, Mariagiovanna Sami, Ettore Speziale, and Michele Tartara. Analyzing the Sensitivity to Faults of Synchronization Primitives. In *DFT*, pages 349–355. IEEE, 2011. ISBN 978-1-4577-1713-0.

[65] Dirk Grunwald and Suvas Vajracharya. Efficient Barriers for Distributed Shared Memory Computers. In Howard Jay Siegel, editor, *IPPS*, pages 604–608. IEEE Computer Society, 1994. ISBN 0-8186-5602-6.

[66] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and Help-first Scheduling Policies for Async-finish Task Parallelism. In *IPDPS*, pages 1–12, 2009.

[67] Andrei Hagiescu, Huynh Phung Huynh, Weng-Fai Wong, and Rick Siow Mong Goh. Automated Architecture-Aware Mapping of Streaming Applications Onto GPUs. In *IPDPS*, pages 467–478, 2011.

[68] Lance Hammond, Vicky Wong, Michael K. Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *ISCA*, pages 102–113. IEEE Computer Society, 2004. ISBN 0-7695-2143-6.

[69] Gilbert Josep Hansen. *Adaptive Systems for the Dynamic Run-time Optimization of Programs*. PhD thesis, 1974.

[70] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.

[71] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[72] Debra Hensgen, Raphael Finkel, and Udi Manber. Two Algorithms for Barrier Synchronization. *Int. J. Parallel Program.*, 17(1):1–17, 1980.

[73] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[74] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, 1993.

[75] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[76] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *PODC*, pages 92–101, 2003.

[77] Antony J. G. Hey and Anne Trefethen. The Data Deluge: An e-Science Perspective. In Fran Berman, Geoffrey C. Fox, and Anthony J. G. Hey, editors, *Grid Computing - Making the Global Infrastructure a Reality*, pages 809–824. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[78] Urs Hölzle and David Ungar. A Third-Generation SELF Implementation: Reconsiling Responsiveness with Performance. In *OOPSLA*, pages 229–243, 1994.

[79] *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, 2008.

[80] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel, 2009.

[81] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *SOSP*, pages 261–276. ACM, 2009. ISBN 978-1-60558-752-3.

[82] Stephen Jenks and Jean-Luc Gaudiot. Exploiting Locality and Tolerating Remote Memory Access Latency Using Thread Migration. *Int. J. Parallel Program.*, 25(4):281–304, 1997. ISSN 0885-7458.

[83] H. Jin and M. Frumkin. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report, NASA, 1999.

[84] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A Model of Computation for MapReduce. In *SODA*, pages 938–948, 2010.

[85] Ralf Karrenberg and Sebastian Hack. Whole-function Vectorization. In *CGO*, pages 141–150, 2011.

[86] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An Analysis of Traces from a Production MapReduce Cluster. In *CCGRID*, pages 94–103. IEEE, 2010.

[87] *The OpenCL Specification, version 1.1.* Khronos OpenCL Working Group, 2010.

[88] Andi Kleen. An NUMA API for Linux, 2004. URL http://www.halobates.de/numaapi3.pdf.

[89] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms.* Addison-Wesley, 1968.

[90] Donald E. Knuth. An Empirical Study of FORTRAN Programs. *Softw., Pract. Exper.*, 1(2):105–133, 1971.

[91] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot^{TM} Client Compiler for Java 6. *TACO*, 5(1), 2008.

[92] Chandra Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the Overhead of Dynamic Compilation. *Softw., Pract. Exper.*, 31(8):717–738, 2001.

[93] Stefan Lankes, Boris Bierbaum, and Thomas Bemmerl. Affinity-On-Next-Touch: An Extension to the Linux Kernel for NUMA Architectures. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *PPAM (1)*, volume 6067 of *LNCS*, pages 576–585. Springer, 2009. ISBN 978-3-642-14389-2.

[94] C. L. Lawson, Richard J. Hanson, D. R. Kincaid, and Fred T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[95] Jaejin Lee, Jungwon Kim, Sangmin Seo, Seungkyun Kim, Jung-Ho Park, Honggyu Kim, Thanh Tuan Dao, Yongjin Cho, Sung Jong Seo, Seung Hak Lee, Seung Mo Cho, Hyo Jung Song, Sang-Bum Suh, and Jong-Deok Choi. An OpenCL Framework for Heterogeneous Multicores with Local Memory. In *PACT*, pages 193–204, 2010.

[96] Jun Lee, Jungwon Kim, Junghyun Kim, Sangmin Seo, and Jaejin Lee. An OpenCL Framework for Homogeneous Manycores with No Hardware Cache Coherence. In *PACT*, pages 56–67, 2011.

[97] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: a Compiler Framework for Automatic Translation

and Optimization. In Daniel A. Reed and Vivek Sarkar, editors, *PPOPP*, pages 101–110. ACM, 2009. ISBN 978-1-60558-397-6.

[98] Stan Lee and Steve Ditko. The Amazing Spider-Man. *Amazing Fantasy*, (15), 1962.

[99] Saeed Maleki, Yaoqing Gao, María Jesús Garzarán, Tommy Wong, and David A. Padua. An Evaluation of Vectorizing Compilers. In *PACT*, pages 372–382, 2011.

[100] Jaydeep Marathe and Frank Mueller. Hardware Profile-guided Automatic Page Placement for ccNUMA Systems. In *PPOPP*, pages 90–99. ACM, 2006. ISBN 1-59593-189-9.

[101] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, Alan J. Miller, and Michael Upton. Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*, 6(1):1–12, 2002.

[102] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, 1992.

[103] *MPI: A Message-Passing Interface Standard, Version 2.2*. Message Passing Interface Forum, 2009.

[104] Markus Mock, Craig Chambers, and Susan J. Eggers. Calpa: a Tool for Automating Selective Dynamic Compilation. In *MICRO*, pages 291–302, 2000.

[105] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *HPCA*, pages 254–265. IEEE Computer Society, 2006.

[106] Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Energy Reduction in Multiprocessor Systems Using Transactional Memory. In Kaushik Roy and Vivek Tiwari, editors, *ISLPED*, pages 331–334. ACM, 2005. ISBN 1-59593-137-6.

[107] Benjamin Moseley, Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. On Scheduling in Map-Reduce and Flow-Shops. In Rajmohan Rajaraman and Friedhelm Meyer auf der Heide, editors, *SPAA*, pages 289–298. ACM, 2011. ISBN 978-1-4503-0743-7.

[108] Matthias S. Müller, G. Matthijs van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C. Brantley, Chris Parrott, Tom Elken, Huiyu Feng, and

Carl Ponder. SPEC MPI2007 - an Application Benchmark Suite for Parallel Systems Using MPI. *Concurrency and Computation: Practice and Experience*, 22(2):191–205, 2010.

[109] Ramachandra Nanjegowda, Oscar Hernandez, Barbara M. Chapman, and Haoqiang Jin. Scalability Evaluation of Barrier Algorithms for OpenMP. In *IWOMP*, pages 42–52, 2009.

[110] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. A Transparent Runtime Data Distribution Engine for OpenMP. *Scientific Programming*, 8(3):143–162, 2000. ISSN 1058-9244.

[111] Dimitrios S. Nikolopoulos, Ernest Artiaga, Eduard Ayguadé, and Jesús Labarta. Scaling Non-regular Shared-memory Codes by Reusing Custom Loop Schedules. *Scientific Programming*, 11(2): 143–158, 2003.

[112] Dorit Nuzman, Sergei Dyshel, Erven Rohou, Ira Rosen, Kevin Williams, David Yuste, Albert Cohen, and Ayal Zaks. Vapor SIMD: Auto-vectorize Once, Run Everywhere. In *CGO*, pages 151–160, 2011.

[113] *The OpenACC^{TM} Application Programming Interface, version 1.0.* OpenACC Standard Committee, 2010.

[114] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *MICRO*, pages 105–118. IEEE Computer Society, 2005. ISBN 0-7695-2440-0.

[115] Balaji Palanisamy, Aameek Singh, Ling Liu, and Bhushan Jain. Purlieus: Locality-aware Resource Allocation for MapReduce in a Cloud. In Scott Lathrop, Jim Costa, and William Kramer, editors, *SC*, page 58. ACM, 2011. ISBN 978-1-4503-0771-0.

[116] Michael Paleczny, Christopher A. Vick, and Cliff Click. The Java HotSpot$^{\text{TM}}$ Server Compiler. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[117] Jongse Park, DaeWoo Lee, Bokyeong Kim, Jaehyuk Huh, and Seungryoul Maeng. Locality-aware Dynamic VM Reconfiguration on MapReduce Clouds. In *HPDC*, pages 27–36, 2012.

[118] Alex Peleg and Weiser Uri. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[119] James Poe, Chang-Burm Cho, and Tao Li. Using Analytical Models to Efficiently Explore Hardware Transactional Memory and Multi-Core Co-Design. In *SBAC-PAD*, pages 159–166. IEEE Computer Society, 2008. ISBN 978-0-7695-3423-7.

[120] Jorda Polo, Claris Castillo, David Carrera, Yolanda Becerra, Ian Whalley, Malgorzata Steinder, Jordi Torres, and Eduard Ayguadé. Resource-Aware Adaptive Scheduling for MapReduce Clusters. In *Middleware*, pages 187–207, 2011.

[121] Constantine D. Polychronopoulos and David J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. Computers*, 36(12):1425–1439, 1987.

[122] Ravi Rajwar, Maurice Herlihy, and Konrad K. Lai. Virtualizing Transactional Memory. In *ISCA*, pages 494–505. IEEE Computer Society, 2005.

[123] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *CGO*, pages 243–254. IEEE Computer Society, 2005. ISBN 0-7695-2298-X.

[124] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC Simulator, 2011. URL http://sesc.sourceforge.net/.

[125] Rice University. High Performance Fortran Language Specification. *SIGPLAN Fortran Forum*, 12(4):1–86, 1993.

[126] Nathan Robertson and Alistair P. Rendell. OpenMP and NUMA Architectures I: Investigating Memory Placement on the SGI Origin 3000. In *International Conference on Computational Science*, volume 2660 of *LNCS*, pages 648–656. Springer, 2003. ISBN 3-540-40197-0.

[127] Richard M. Russell. The Cray-1 Computer System. *Commun. ACM*, 21(1):63–72, 1978.

[128] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. *X10 Language Specification, version 2.2*. IBM, 2012.

[129] Vivek Sarkar. Optimized Unrolling of Nested Loops. In *ICS*, pages 153–166, 2000.

[130] Nir Shavit and Dan Touitou. Software Transactional Memory. In *PODC*, pages 204–213, 1995.

[131] Jun Shirako and Vivek Sarkar. Hierarchical Phaser for Scalable Synchronization and Reductions in Dynamic Parallelism. In *IPDPS*, 2010.

[132] Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phaser Accumulators: A New Reduction Construct for Dynamic Parallelism. In *IPDPS*, pages 1–12, 2009.

[133] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-memory Multicore Systems. In *SC*, 2009.

[134] Ettore Speziale, Andrea Di Biagio, and Giovanni Agosta. An Optimized Reduction Design to Minimize Atomic Operations in Shared Memory Multiprocessors. In *IPDPS Workshops*, pages 1300–1309. IEEE, 2011. ISBN 978-1-61284-425-1.

[135] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen mei W. Hwu. Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs. In *CGO*, pages 111–119, 2010.

[136] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation*. Prentice Hall, third edition, 2006.

[137] Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. Data and Thread Affinity in OpenMP Programs. In *MAW '08: Proceedings of the 2008 workshop on Memory access on future processors*, pages 377–384. ACM, 2008. ISBN 978-1-60558-091-3.

[138] The Apache Software Foundation. Hadoop MapReduce. http://hadoop.apache.org/mapreduce, Feb 2012.

[139] William Thies and Saman P. Amarasinghe. An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design. In Valentina Salapura, Michael Gschwind, and Jens Knoop, editors, *PACT*, pages 365–376. ACM, 2010. ISBN 978-1-4503-0178-7.

[140] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Using Hardware Counters to Automatically Improve Memory Performance. In *SC*, page 46. IEEE Computer Society, 2004. ISBN 0-7695-2153-3.

[141] Robert M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1), 1967.

[142] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[143] *UPC Language Specifications, version 1.2*. UPC Consortium, 2005.

[144] Hans Vandierendonck, George Tzenakis, and Dimitrios S. Nikolopoulos. A Unified Scheduler for Recursive and Task Dataflow Parallelism. In Lawrence Rauchwerger and Vivek Sarkar, editors, *PACT*, pages 1–11. IEEE Computer Society, 2011. ISBN 978-1-4577-1794-9.

[145] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Resource Provisioning Framework for MapReduce Jobs with Performance Goals. In *Middleware*, pages 165–186, 2011.

[146] John von Neumann. First Draft of a Report on the EDVAC. *Annals of the History of Computing, IEEE*, 15(4), 1993.

[147] David W. Wall. Limits of Instruction-Level Parallelism. In David A. Patterson, editor, *ASPLOS*, pages 176–188. ACM Press, 1991. ISBN 0-89791-380-9.

[148] Cheng Wang, Ho-Seop Kim, Youfeng Wu, and Victor Ying. Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. In *CGO*, pages 244–258. IEEE Computer Society, 2007. ISBN 978-0-7695-2764-2.

[149] Joel L. Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, and Andrey Balmin. FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads. In Indranil Gupta and Cecilia Mascolo, editors, *Middleware*, volume 6452 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2010. ISBN 978-3-642-16954-0.

[150] William A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1), 1995.

[151] Tse-Yu Yeh and Yale N. Patt. Two-Level Adaptive Training Branch Prediction. In *MICRO*, pages 51–61, 1991.

[152] Marcelo Yuffe, Ernest Knoll, Moty Mehalel, Joseph Shor, and Tsvika Kurts. A Fully Integrated Multi-CPU, GPU and Memory Controller 32nm Processor. In *ISSCC*, pages 264–266. IEEE, 2011. ISBN 978-1-61284-303-2.

[153] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In Richard Draves and Robbert van Renesse, editors, *OSDI*, pages 29–42. USENIX Association, 2008. ISBN 978-1-931971-65-2.

[154] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: a Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, pages 265–278, 2010.