An Empirical Study of Program Performance of OpenMP Applications on Multicore Platforms

Abdelhafid Mazouz

UNIVERSITÉ de VERSAILLES SAINT-QUENTIN-EN-YVELINES

Thèse

pour obtenir le grade de

Docteur de l'Université de Versailles Saint-Quentin-en-Yvelines

Discipline:

INFORMATIQUE

Présentée et soutenue publiquement par

Abdelhafid MAZOUZ

Le 11 décembre 2012

Sujet de la thèse:

Une étude empirique des performances des applications OpenMP sur les plateformes multi-cœurs

An Empirical Study of Program Performance of OpenMP Applications on Multicore Platforms

Jury:

Pr.	Denis Barthou	Université de Bordeaux	Directeur
Pr.	Sid Touati	Université de Nice Sophia Antipolis	Co-directeur
Pr.	Jean-François Méhaut	Université de Joseph Fourier de Grenoble	Rapporteur
Pr.	Arndt Bode	Université Technique de Munich	Rapporteur
Pr.	William Jalby	Université de Versailles Saint-Quentin-en-Yvelines	Examinateur
Pr.	Claude Timsit	Université de Versailles Saint-Quentin-en-Yvelines	Examinateur
Dr.	Raphaël David	Commissariat à l'Énergie Atomique	Examinateur

Thèse préparée au sein du laboratoire PRiSM

Remerciements

Tout d'abord, je souhaiterais dédier cette thèse à mes parents, sans qui celle-ci n'aurait probablement pas eu lieu. Je tiens vraiment à les remercier pour leur soutien et leurs encouragements.

Je tiens aussi à remercier mes deux encadrants Sid TOUATI et Denis BARTHOU pour la qualité de leur encadrement, encouragements, gentillesse et disponibilité permanente, une grande chance et un vrai plaisir de travailler avec eux. Une pensée particulière pour Sid, avec qui j'ai commencé mon stage de master, ayant découvert à ces cotés le monde passionnant du HPC.

Je remercie également les membres de mon jury, William JALBY, Claude TIMISIT, Raphael DAVID, et tout particulièrement Arndt BODE et Jean-Franǫis Méhaut pour le temps qu'ils m'ont consacré, leurs critiques constructives et la qualité de leurs remarques qui m'ont été d'une grande utilité.

Un grand merci aux membres du département d'informatique de l'université de Munich, où j'ai passé deux mois qui m'ont étè vraiment bénéfiques; Notamment Josef Weindendorfer et Perta Piochacz pour leur accueil et leur disponibilité.

Sans oublier bien sûr mes collègues au sein du laboratoire PRiSM avec qui j'ai passé d'agréables moments tout au long du déroulement de ma thèse.

A mes parents.

Résumé

Les architectures des machines multi-cœurs actuelles deviennent de plus en plus complexes à cause du modèle de conception hiérarchique adopté. En effet, dans ces machines, les coeurs partagent plusieurs ressources comme des mémoires cache, des bus mémoires, des pré-chargeurs ou des nœuds mémoires. Par conséquent, atteindre de très bonnes performances sur ces machines nécessite une compréhension approfondie des interactions qui existent entre les applications multi-threads et le matériel sous-jacent. Une compréhension précise de ces interactions aidera pour mieux estimer les vraies performances de ces programmes d'un coté, et de fournir des techniques d'optimisation de programmes efficaces de l'autre coté. Dans cette thèse, nous étudions deux aspects importants pour les performances des applications multi-threads. Nous montrons que la stabilité des performances est un critère important à considérer dans le processus d'évaluation des performances, et que le placement des threads est une technique efficace pour améliorer les performances des programmes d'un coté et pour une meilleure stabilité des performances de l'autre coté.

À cause des interactions qui existent entre les couches logicielles et le matériel, les temps d'exécution des programmes peuvent être instables. En réalité, lancer un programme plusieurs fois peut engendrer plusieurs temps d'exécution, produisant ainsi, ce qui appelé une variabilité dans les temps d'exécution des programmes ou variabilité des performances. Plus cette variabilité est importante, plus le risque de surestimer ou de sous-estimer le vrai comportement du programme est élevé. Donc ces variations peuvent conduire à de conclusions erronéses sur les performances d'un programme. Pour étudier ces variations, nous proposons une évaluation statistique rigoureuse des performances des applications multi-threads. En utilisant des configurations expérimentales fixes, notre objectif est de 1) quantifier ces variations, et 2) identifier les facteurs qui influenceent la variabilité des temps d'exécution.

En considérant les architectures multi-cœurs avec des mémoires cache partagées, notre étude de la variabilité des temps d'exécution nous a permis de constater que ces applications sont sensibles au placement ou affinité des threads. L'affinité des threads est aussi apparu comme l'un des moyens les plus importants pour accélérer le temps d'exécution des programmes. Cependant, il est encore difficile de comprendre comment l'affinité des threads peut contribuer à améliorer ou à détériorer les performances. Pour ce faire, nous avons pris le partage des données entre threads comme métrique, et en utilisant une méthode de profilage, nous étudions les performances de plusieurs stratégies de placement des threads en termes de stabilité et d'amélioration des performances.

En dernier, les applications OpenMP peuvent avoir plusieurs régions parallèles où chacune peut avoir un différent modèle de partage des données entre threads. Ceci implique qu'il est très rare que la même stratégie de placement des threads puisse avoir les meilleurs performances pour toutes les régions. Pour cette raison, nous proposons une approche qui autorise les migrations de threads. En effet, nous proposons d'instrumenter les programmes OpenMP dans le but d'identifier les régions parallèles, puis de calculer une affinité différente pour chaque région parallèle. Nous donnons aussi une analyse qui permet d'identifier les conditions nécessaires pour le bon fonctionnement de cette approche.

Mots clés: OpenMP, partage de données, localité de données, affinité entre threads, multi-cœurs, parallélisme, évaluation des performances.

Abstract

Current architectures of multicore machines are becoming increasingly complex due to hierarchical designs where multiple cores share common resources like caches, memory buses, prefetchers or memory nodes. Consequently, extracting high performance from these machines requires a deep understanding of the interactions between multi-threaded applications and the underlying hardware. An accurate understanding of these interactions helps to better estimate the true program performance behaviour of multi-threaded applications in one side, and to provide efficient program optimisation techniques on the other side. In this thesis, we study two important aspects for the performance of multi-threaded applications. We show that performance stability is an important criteria to consider in the process of performance evaluation, and thread placement is an effective technique for improving program performance in one hand, and for a better performance stability on the hand.

Due to the interactions between the software layers and the hardware, program execution times may be instable. In fact, running a program multiple times can lead to distinct program execution times, thus producing what is called a variability of program execution times or variability of program performance. The more this variability is important, the more the risk of overestimating or underestimating the true program performance is high. Consequently, these variations can lead misleading conclusions about program performance. To study these variations, we use a rigorous statistical performance evaluation of multi-threaded applications. Using fixed experimental setups, we aim to 1) quantify these variations, and 2) identify the factors that influence the variability of program execution times.

Regarding multicore architectures with shared caches, our study of the variability of program execution times allowed us to identify that these applications are sensitive to thread affinity. Thread affinity has also appeared to be one of the most important factors to accelerate program execution times. However, it is still unclear how thread affinity contributes to increase or to decrease execution times. Taking the inter-thread data sharing as metric, and using a profile guided method, we investigate the performance of multiple thread affinity strategies in terms of performance stability and performance improvement.

Last, OpenMP applications may have multiple parallel regions, where each region may have a distinct inter-thread data sharing pattern. It is unlikely that a single thread affinity produces the best program performance for all the parallel regions. For this reason, we propose an approach that allows thread migrations. Indeed, we instrument OpenMP programs in order to identify OpenMP regions, then we compute a distinct thread affinity for each parallel region. Furthermore, we provide an analysis about the required conditions to improve the effectiveness of the approach.

Keywords: OpenMP, thread affinity, data sharing, data locality, multicore processors, parallelism, performance evaluation.

Contents

Chapter 1

Introduction

High performance computing refers to running applications with high needs in terms of computational power and having large data input sizes. These applications cover areas such image processing, weather modeling, financial analyses or computational fluid dynamics simulations. To satisfy this increasing demand for power processing, computer architecture has made incredible advancements in processing hardware thanks to techniques like high cpu clock frequency, deeper processor pipelines, using larger caches, or multiple functional units at the micro-architectural level on one hand and using multiple processors at the architectural level on the other hand. Moreover, advances in integrated circuits technology allow to pack more and more independent processing units in a single die resulting in the emergence of what is called multicore technology.

The increasing number of processing units in today's multicore processor architectures allows to run multiple independent applications concurrently. Moreover, each application may run with multiple threads, hence, improve overall application performance by exploiting thread level parallelism. Unfortunately, the increasing architectural complexity of these new state of the art designs makes the task of achieving the peak performance non-trivial. Consequently, a better understanding of the interactions between the operating system layers, the applications and the underlying hardware platforms is of high importance. In fact, due to these interactions, program performance may not be stable. Therefore, multiple runs of an application may produce multiple program performance behaviours, also called performance variations or performance instability. Depending on execution environments, performance variations can be small or large. Larger variations make the process of accurately determining the performance of a program more challenging, because the risk of overestimating or underestimating the true performance behaviour of a program is high. So, the ability to characterise and to quantify those interactions can be useful in the process of performance evaluation and analysis, compiler optimisation techniques and operating system job scheduling allowing to achieve better performance stability, reproducibility and predictability.

Multicore processor architectures have multiple shared resources such as common buses, last level prefetchers, a hierarchy of caches or memory nodes creating complex topologies. Running multiple parallel or concurrent applications on top of these platforms requires adequate parallelisation strategies to take benefit from the available resources on one side, and intelligent operating system scheduling policies that carefully allocate shared resources on the other side. Indeed, naive or inadequate resources sharing by multiple threads or processes can generate resource contention, therefore leading to severe overall performance degradation. In this context, to be closer to the theoretical peak performance, it is important to extract the communication/data sharing patterns exhibited by parallel applications and place them accordingly

onto the hardware architecture topology. Several techniques have been developed to tackle this problem. We can mainly consider three cases: 1) manually by the application programmer, 2) compilers and 3) operating systems or runtime libraries. Our aim in this thesis is to study the interactions of parallel OpenMP applications and the hardware platforms from the performance stability perspective on one hand, and to study the impact of thread placement on the overall program performance on the other hand.

1.1 Goals and contributions of the thesis

The contribution of this thesis is a study of program performance of OpenMP applications on multicore processors. Applications are studied from both the performance stability on one hand and the relation between the inter-thread data sharing and thread placement techniques on the other hand.

1.1.1 Performance stability of OpenMP applications

The first part of this thesis studies the variability of program execution times as a performance instability metric in native executions of OpenMP programs. Underestimating this variability can very likely affect the accuracy of any program performance study, and at worst can lead to misleading conclusions about the true performance behaviour of the application. In this context, an accurate quantification and qualification of this variability is important. We focus on the task of isolating the factors that may influence the most this performance variability.

First, we give a definition of variability of program execution times and present a rigorous performance evaluation methodology for better performance reproducibility. Second, we perform multiple experiments that aim to measure the variability of program execution times. By fixing the experimental setup, we stress various micro-architectural, application, and operating system components. Our goal is to understand the influence and the sensitivity of OpenMP programs to each of these components and their direct relation on program execution times variability.

1.1.2 Enhancing data sharing with efficient thread placements

Our effort to isolate the factors that contribute to increase the variability of program execution times, has led us to find that OpenMP applications are very sensitive to thread placement. Indeed, bad threads placement can lead to non-negligible performance variability. Thread placement of OpenMP threads mainly impacts cache performance and may exacerbate non-uniform memory access effects. Therefore, knowing the influence of thread placement on performance improvement or variability is necessary. In this context we conduct the studies presented in the second part of this thesis. By considering the inter-thread data sharing as a metric for thread placement, our objective is to understand the impact of multiple thread placements on application performance for a given OpenMP application. In fact, we do not track inter-thread data sharing, but inter-thread memory cache lines sharing. However, in the remainder of this thesis, we use the terms cache memory lines, data sharing or data reuse without distinction.

For this study, we proceed in two steps. First, we perform an evaluation and analysis of multiple thread placements strategies on some multicore architectures featuring different hierarchies of shared caches. The tested strategies are *application-wide*, this means that we fix the same thread placement from the beginning of an application until it finishes its execution. Moreover,

we decompose the tested strategies into two classes: 1) strategies that do not require information about the characteristics of the application, and 2) strategies that do require information about the characteristics of the application.

Second, we consider that any parallel program may have different phases, each with a distinct performance behaviour. Each phase is also characterized with a temporal window (duration of the phase). It is possible to identify program phases using metrics such cache miss ratio, instructions per cycle, or the amount of data sharing between threads. A program phase may have different granularities: going from few instructions, to function calls. Thus, it is unlikely that the same fixed thread placement gives the best performance for all parallel phases. Indeed, if each parallel phase exhibits a distinct sharing pattern between threads, then an application-wide thread affinity strategy will not be able to effectively exploit that distinct patterns. This situation has motivated us to extend our study to account for multiple OpenMP phases. In this thesis, we consider that program phases in OpenMP programs correspond to OpenMP parallel regions. Consequently, we investigate other thread placement solutions based on thread migrations or per-phases thread placement.

1.2 Dissertation outline

The outline of this thesis is organised as follows. Chapter 2 reminds the evolution of parallel machine architectures used nowadays and the most common parallel programing paradigms used to exploit the power of these parallel machines. Chapter 3 first introduces the problem of program execution times variability. After that, it gives a large overview about the problems of measuring data locality and the impact of shared cache access contention. Last, it presents techniques to exploit data sharing using thread placement in multicore processors. Chapter 4 presents an experimental study for measuring and analysing the variability of program execution times. Chapter 5 discusses the relation between the inter-thread data sharing using optimised thread placement techniques and the performance stability. Chapter 6 extends Chapter 5 and presents an approach to compute efficient thread placement techniques for multiple OpenMP parallel phases. Finally, Chapter 7 concludes this thesis and gives some future research proposals.

Chapter 2

The Multicore Era

In this chapter, we give a short overview of the evolution of current computer systems from both the hardware and software perspectives. First, we revisit hardware implemented techniques that aim to increase parallelism and performance. We distinguish between micro-architectural optimisations inside a single processing unit (not visible to the programmer/compiler) and architectural optimisations (visible to the programmer/compiler) which combine multiple processing units. Second, we present some parallel programming models, languages and runtimes for efficient and effective parallel machines utilisation.

2.1 Hardware evolution: the race for more parallelism

Computer technology has made an incredible progress since early days of general purpose computing. Advances in integrated circuits technology and architecture design have allowed to build machines going from single to multiple processing units per chip. To improve performance, computer designs have focused on increasing parallelism by executing multiple streams of instructions. In this quest for more parallelism, it is possible to say that hardware performance optimisation is achieved by means of two complementary approaches. With the first approach, the hardware optimisations logic is implemented inside a single processing unit. Mainly, they are intended to enhance the single thread performance. From the programming perspective, these optimisations are hidden or not visible to the programmer or to the compiler. We can call them as micro-architectural optimisation. The main goal of micro-architectural optimisations is to lower the execution latency of a single instruction, while increasing the number of instructions that can be executed in a single clock tick. Many optimisations such as deeper pipelines, better branch predictors, larger cache sizes, etc. are an example. Regarding the second approach, we can talk about architectural optimisations. From the programming perspective, an architectural optimisation is exposed and is visible by the programmer and/or by the compiler. The goal of this approach is to increase the throughput by executing multiple independent instructions streams on multiple processors in parallel. Thus, increasing parallelism. This section presents some concepts regarding the design of parallel machines.

2.1.1 Taxonomy of parallel machines

According to several criteria, there are different ways to classify parallel machines. The most widespread classification is called the Flynn's Taxonomy [Fly72]. Table 2.1 summarises this classification.

- **SISD** (Single Instruction, Single Data). This organisation represents traditional sequential machines. A processing unit executes instructions from one single stream sequentially,

	Data Stream	
Instruction Stream	Single Instruction, Single Data	Single Instruction, Multiple Data
	Multiple Instruction, Single Data	Multiple Instruction, Multiple Data

Table 2.1: Flynn's Taxonomy

	Communication/Synchronisation	
Memory organisation	Global Memory, Shared Variables	Global Memory, Message Passing
	Distributed Memory, Shared Variables	Distributed Memory, Message Passing

Table 2.2: Johnson's MIMD classification

the execution of instructions may be overlapped within the pipeline's stages. An instruction operates only on one data stream during any clock cycle. Some embedded machines belong to this architecture organisation.

- **SIMD** (Single Instruction, Multiple Data). In this class of architectures, all the processing units execute the same instruction at the same time and synchronise. However, the different processing units operate on different data. Vector processors belong to the SIMD architectures class.

- **MISD** (Multiple Instruction, Single Data). This class represents architectures where a single data stream is fed into multiple processing units. Each processing unit operates on the data independently by acting with different instructions streams. However, even if all the units operate on the same data stream, each unit operates on a distinct data. This category includes specialised machines like `systolic` machines where processing units are arranged given some fixed topology, are highly synchronised and which are supported by a generalised processor.

- **MIMD** (Multiple Instruction, Multiple Data). This class represents multiprocessor machines where, each processor executes its own code independently and asynchronously from others. Each processor operates on its own input data stream. Processors communicate data among them. Most current supercomputers, networked parallel machines and multiprocessors follow this architecture organisation.

Flynn's classification suffers from some limitations. The MIMD class for instance includes a wide variety of computers. For this reason Johenson [Joh88] proposed further classification of such machines, it is based on their memory organisation (*global/distributed*) and the mechanism used for communications/synchronisation (*shared variables/message passing*). Table 2.2 summarises this classification.

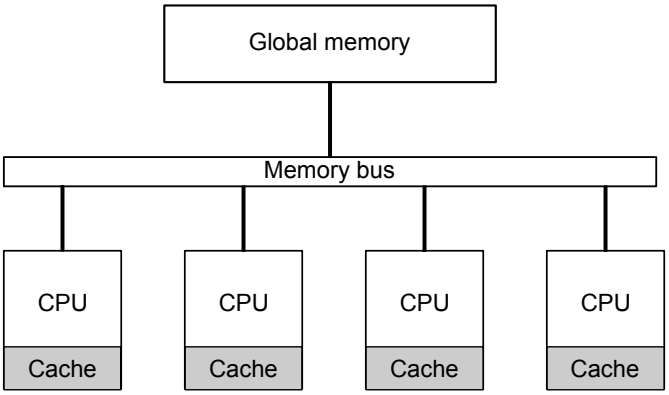


Figure 2.1: A GMSV machine

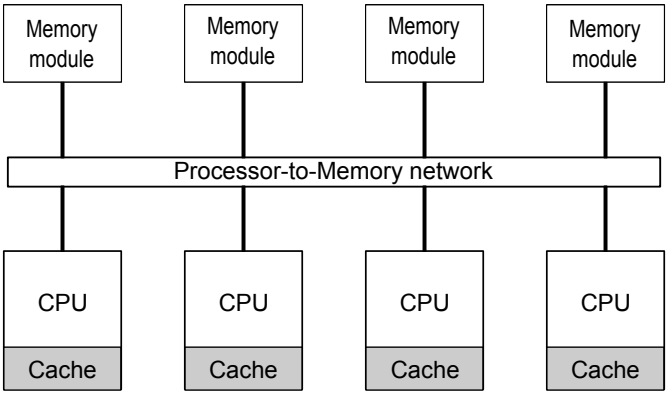


Figure 2.2: A DMSV machine

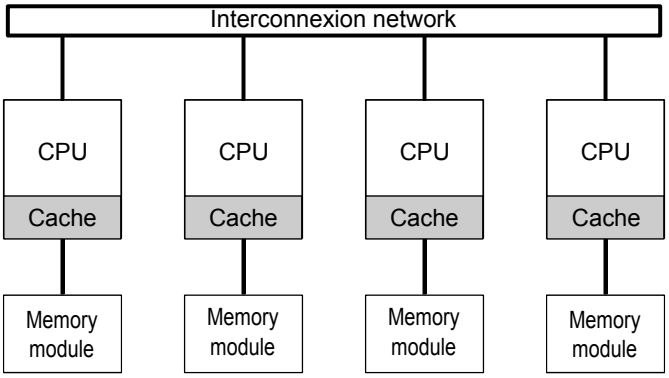


Figure 2.3: A DMMP machine

- **GMSV** (Global Memory, Shared Variables). This class represents traditional shared-memory multiprocessors also known as `symmetrical multiprocessing (SMP)` (see Figure 2.1). A set of identical processors are connected by a bus or a cross bar switch and access a single global memory with an equal time latency. We can talk about `Uniform Memory Access (UMA)`. If processors feature memory caches which is the case nowadays, we talk about `Cache-Coherent UMA`. The coherency is ensured by protocols implemented within each memory cache controller [CSG98].

- **GMMP** (Global Memory, Message Passing). This class represents machines implementing a global addressing space and where communications are achieved by means of message

passing instead of using shared variables. This class of architectures is not widely used.

- **DMSV** (Distributed Memory, Shared Variables). This class represents machines where the memory is physically distributed, but all the processors have the same shared address space allowing remote access of data. Regarding the performance of memory operations, the latency depends on the data location (see Figure 2.2). In general, accessing local data is faster than accessing remote data. In this case we talk about `Non-Uniform Memory Access (NUMA)` machines. Again, if processors feature memory caches with an implemented coherency protocol among them, we can talk about `Cache-Coherent NUMA` machines.

- **DMMP** (Distributed Memory, Message Passing). Also known as `distributed-memory multi-computers` (see Figure 2.3). This class of architectures implements a model where the memory is physically distributed without any possibility to access remote data transparently. Data communication is achieved explicitly by message passing through a communication network.

2.1.2 Instruction-level parallelism

Since the mid-80s, almost all processor architectures rely on the *pipelining* principle to overlap the execution of multiple instructions and improve performance. Practically, executing an instruction requires to go through some intermediate stages or basic operations before a result is produced. For example, we can say that a pipeline has four stages:

1. Instruction fetch (IF): bring the instruction from memory;
2. Instruction decode (ID): decide which operation to execute;
3. Instruction execute (EX): use the arithmetic and logic unit to evaluate and produce a result. If required bring the operands from memory;
4. Write result (WR): store data back to memory.

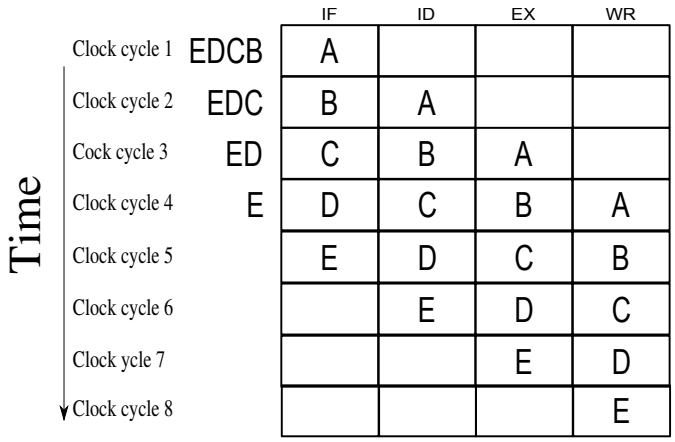


Figure 2.4: The instruction flow for five instructions (A,B,C,D,E) in a pipeline with four stages

If we consider all the steps above as an atomic operation or a black box, the execution of each instruction has to wait until all these operations are finished for a previously issued instruction. This execution model can generate a non negligible overhead. In addition of splitting-up the execution of an instruction using intermediate stages, in the *pipelining* principle, each stage

involves a separate and an independent hardware component. This design allows theoretically, to issue an instruction to execute as soon as the first stage in the pipeline becomes free. It is then possible to execute multiple instructions in parallel from one instructions stream leading to better program performance. The principle of overlapping instructions in the pipeline to increase performance is called the *Instruction-Level Parallelism* (ILP). Figure 2.4 reports the case where up to four instructions are executed in parallel with a pipeline of four stages. The time that each instruction spends in each stage of the pipeline is controlled by the clock frequency or the frequency of operation; that means, a given stage requires one or many cycles to perform the operation. The higher the clock frequency, the lower the time spent on the pipeline stages.

To achieve a higher clock frequency, architecture designers have to increase the depth of the pipelines. With this design, pipeline stages operate on simpler operations with a low overhead. However, even with that design, there are some limitations to keep all the pipeline stages busy. Keeping pipeline stages busy all the time is quite difficult. There are multiple reasons for that: a non-fixed execution time for some complex instructions[1], cache and/or memory effects, etc. When such instructions are issued, they are taking long periods of time inside the different stages of the pipeline. Therefore, if new instructions have to pass through the same stages, the pipeline needs to suspend their execution until the required resources become free. And of course, it contradicts the principle of issuing one instruction in the pipeline each cycle. To overcome this limitation, computer architects made the choice of duplicating some functional units and hardware components involved in the different stages of the pipeline. This duplication aims to reduce the impact of such bottlenecks by allowing the execution of multiple instruction in parallel per cycle. With the advent of this design, we can talk about *superscalar* processors (Alpha 21264, HP PA 8500 or Pentium III/4 [Joh02]). Data dependency between consecutive instructions can also be a reason for suspending the execution of an instruction in the pipeline. It happens when an instruction depends on the result produced by another instruction. In this situation, the second instruction has to wait until the first one finishes its execution and produce the required result. Many other hardware optimisation techniques are implemented inside the pipeline logic as: out of order execution, branch prediction, rename registers, etc. All these techniques contribute to increase the instruction-level parallelism [Joh02].

2.1.3 SIMD parallelism and vector processors

There are multiple applications where a single operation has to be applied on large sets of data elements. For instance, it is possible to consider matrix-oriented computations and media/sound processing. In programming languages, loops represent an abstraction of repetitive operations. If the different iterations of the loop do not carry any or little dependencies, and if they operate on large data items, it is then possible to increase the amount of parallelism among the iterations of the loop. This parallelism is inherent to the large data set.

Listing 2.1: A simple loop sample

```
for (i=0; i<N; i++)
    X[i] = Y[i] + Z[i]
```

Listing 2.1 shows an example of a loop where the iterations do not carry any data dependency. It means that each iteration can be executed independently and in parallel, and each executing for a distinct data item. This execution model allows to expose what is called *Data-Level Parallelism*. One way to exploit this parallelism is to use vector or SIMD instructions.

[1] This is actually the case in CISC architectures compared to RISC architectures

Basically, one vector instruction operates on a collection of data items in parallel. If we look to Listing 2.1 again, using vector instructions may translate to the use of up to four instructions: two instructions to load vectors Y and Z, one instruction to perform the addition and finally one instruction to store back the vector X. A vector operation has the property that all the functional units are pipelined. Although, the instruction operates on a whole vector, the operations are performed sequentially on the consecutive elements of a vector inside these pipelined units.

There are two main types of architectures of vector processors [Joh02]:

1. **Vector-register processor**. All the vector operations (except load and store) are performed between the different registers of the machine.
2. **Memory-memory processor**. In this class of vector architectures, all the vector operations are memory to memory operations.

Besides vector processors, we can find multimedia extensions to standard instructions sets (in micro-processors) which belongs also to the SIMD execution model. For X86 architectures, Intel introduced in 1996 the `MMX` (Multi Media eXtensions) instruction set which operates on 64-bits floating-point registers. The later was extended later by the `SSE` (Streaming SIMD Extensions) in order to operate on 128-bits wide registers. In 2010, Intel added the `AVX` (Advanced Vector Extensions) that doubles again the registers width to 256 bits. For non-X86 architectures, we can think to the `AltiVec` (128-bits wide registers) technology from IBM, Motorola and Apple for `PowerPC` processors. Unlike vector processors (a register can hold up to 128 64-bit elements: Cray T-90), the relative small register size of multimedia SIMD extensions can be considered as short-vector SIMD processors.

2.1.4 Multiprocessor parallelism

In the previous section, we discussed some hardware techniques to exploit parallelism within a single stream of instructions. Although this approach is effective for programs with a high amount of *fine-grain* parallelism, programs may implement high-level parallelism that is limited or hard to exploit automatically by the hardware logic. This is true for instance for programs which implement *medium-grain* or *coarse-grain* parallelism. As an example of this high-level parallelism, we can consider the case of *Thread-Level Parallelism*. It is structured as separate and independent streams of instructions also called threads of execution. Moreover, thread-level parallelism can be exposed explicitly by creating multiple threads of execution at the high-level programming language. This is still true, because there are many applications which are inherently parallel. For example, an online file server receives file requests, each request to the server can be processed independently and in parallel.

For the purpose of exploiting thread-level parallelism, advances in computer design technology allowed computer architects to start building parallel machines at the architectural level. Unlike instruction-level parallelism which exploit parallelism at the micro-architectural level (within a processor), taking benefit from thread-level parallelism requires to aggregate multiple processing units, processors or microprocessors within a single machine and ensures that they operate in parallel in a coherent way.

2.1.4.1 Multicore processor architecture

Thanks to advances in integrated circuits manufacturing, processor performance has significantly increased since their first introduction to market in early 70s. In 1975, Gordon Moore

described what is called nowadays as the *Moore's law*. The law states that the number of transistors that can be placed on an integrated circuit would double every couple of years [Moo75] with the same surface, for the same price. This law is actually a revising of his earlier prediction in 1965. While this technology trend led to smaller, faster and higher number of transistors, it has contributed to get higher processor clock frequency and allowed multiple micro-architectural optimisations which aim to increase the ILP.

Due to various physical limitations such as power dissipation and signal propagation delays, building high performance processors with the increase of clock frequency only, is not anymore a viable solution. While clock frequency does not increase, the build process of processors continues to make advances by shrinking the die size. Moore's law has nothing to do with the expected performance, it predicts only the number of transistors to put on a single die. So, a question can rise: how to use the new die space? Due to the hard task of exploiting the ILP, it is useless to add more transistors within a single processor. On the other hand, there are many applications which expose thread-level parallelism. These observations led computer architects to propose designs where a set of identical processors (*cores*) are put together within a single die or a chip. This design is called now *multicore* processors. A multicore processor consists of multiple and identical cores on a single chip, all the cores share the same micro-architectural optimisations. Most often, multicore processors share one or several levels of memory caches. Multicore processors are also known as *Chip multiprocessors* (CMPs) for SMP on a chip, because they are sharing many architectural features design with SMPs.

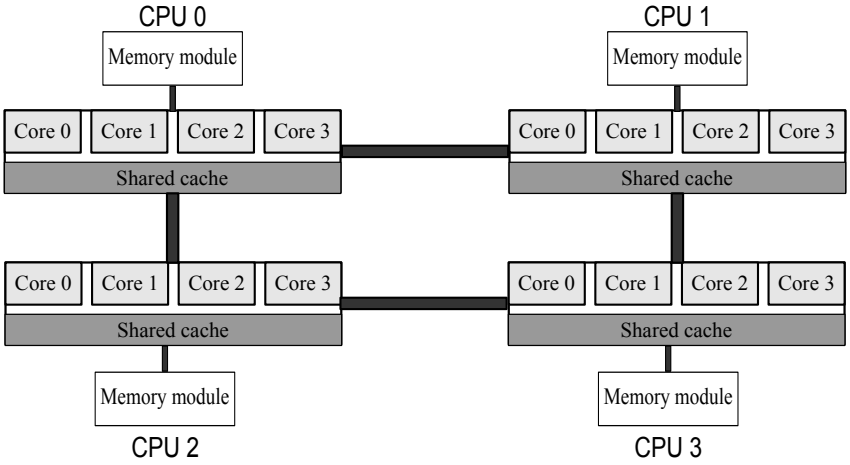


Figure 2.5: A NUMA machine with 4 multicore processors

The growing gap between processor performance and memory performance has led manufacturers to propose highly hierarchical machines to alleviate this problem. The common architectural design consists of two or more cores sharing some levels of memory caches. When building parallel machines with multiple multicore processors, a simple design would be to connect all the processors to a single shared memory through a bus topology (as the traditional SMP design). Bus snooping technique has the drawback of achieving poor scalability (weak scalability of bus snooping coherency protocols). So, to achieve good performance, scalability increase of new multicore-based shared memory machines is of paramount importance. To overcome this limitation, new trends propose distributed shared memory architectures where, each multicore processor is attached to its own main memory and where all the processors are connected with proprietary interconnect networks (`Quickpath` from `Intel` and `Hyper-Transport`

from AMD for instance). When a core in one processor requires data located on a remote processor, it has to send a request through the network. It is clear from this design that the cost of accessing the local or the remote data is not the same, we can talk again about NUMA machines (see Figure 2.5). Consequently, an effective use of such machines whether by programmers, compilers or runtime systems is more challenging.

2.2 Parallel programming models, languages and runtimes for parallel machines

In this section, we review most popular parallel programing paradigms and some specific implementations. Among the variety of parallel programming models, we mainly discuss four parallel programming models that are widely used in the area of parallel programming:

- Shared memory programming model.
- Message passing programming model.
- Virtual shared memory programming model.
- Hybrid programming model.

The programming models that we mention earlier represent abstractions of the machine structure. Independently of the machine implementation, it is also possible to define two high-level programming models:

- **SPMD** (Single Program, Multiple Data). All the tasks of the parallel program execute their copy of the same program simultaneously. The set of tasks operate on different input data. It is not necessary that all the tasks execute the same stream of instructions, logic can be added inside the program to allow tasks to execute only a sub-set from the program. This model is very common in the community of high performance computing.
- **MPMD** (Multiple Program, Multiple Data). All the tasks of the parallel program execute different programs or instructions streams simultaneously. The programs can be threads, message passing, data parallel or hybrid. Each task operates on a distinct input stream. An example of applications implementing this model is the Client/Server applications.

Before addressing the most used programming paradigms, it is important to know the different possibilities to leverage the power of parallel machines. We mainly have two implementation possibilities of parallelism: 1) languages or language extensions/pragmas and 2) using libraries.

2.2.1 Message passing programming

In this programming model, a program creates a set of parallel tasks, each is restricted to its own memory location. This gives the possibility to collocate multiple tasks on the same physical machine or across multiple machines. Data communication between tasks is explicit by sending and receiving messages through a message passing protocol. Data transfer requires cooperative operations to be performed by each task (a send must have a receive operation). Although this programming model highly involves the responsibility of the programmer (all the parallel operations have to be specified explicitly), it offers better control on a parallel program. There are mainly two important implementations of this programming model: PVM and MPI. We limit our discussion to MPI which is presented in the following paragraph.

Message Passing Interface (MPI)
MPI [MPI] is a high-level API for parallel programming which implements the message passing paradigm. Before the MPI standard, every parallel system vendor provided its own version of message passing model. Due to the variety of implementations, it was a hard task to port an application from one parallel system to another. To tackle this issue, a group of parallel computer vendors, university researchers and software developers proposed a standard and portable interface. Although, the high-level interface is common, each hardware vendor may implement an optimised MPI version for their own machines and allows to take benefit from specific network topologies. The MPI library is defined for both Fortran and C/C++ languages. Besides the message passing model, MPI uses the SPMD model, where an MPI application creates a set of processes, each executing a copy of the program and using its own data input. MPI is used to program a variety of MIMD machines going from `massively-parallel` machines to clusters of workstations.

2.2.2 Shared memory programming

In this programming model, multiple tasks share a global address space in which they read and write to asynchronously. From the programmer's point of view, there is no notion of explicit data communication or data exchange between tasks. Thanks to this principle, the programming effort can be greatly simplified, it is all about accessing and modifying memory locations in the shared address space. For parallel machines with memory caches, cache coherency protocols play an important role on keeping all the copies of data in different locations in sync. However, cache coherency protocols in stand-alone are not able to prevent concurrent access to shared data. Therefore, to ensure the coherency of shared objects, explicit synchronisation mechanisms such as locks and semaphores are required to control concurrent accesses.

2.2.2.1 Libraries for parallel programming in shared memory machines

We discuss in this section thread libraries and TBB.

Lightweight processes (Threads)
A thread-model is an approach to achieve parallelism in shared-memory parallel systems. A single process can have multiple flows of control, called threads of execution. These threads share the same global memory (code and data segments, heap) but, each thread has its own stack and program counter. Thread libraries provide functions for thread management such as creation, control, termination and synchronisation. Writing parallel or concurrent programs using thread programming requires more care, it is often associated to low-level OS programming where specific skills are needed. However, thread programming is a powerful way to achieve high performance in shared memory systems. Thanks to the rich API and resources offered to the programmer, programming with threads allows better control on the application behaviour. As usual, due to portability issues of different multi-threading libraries, the `POSIX Threads`[2] (also referred as `Pthreads`) standard [Thea] was proposed for a standardised programming interface. Nowadays, implementations of the `Pthreads` API are available on many `POSIX`-conformant operating systems.

Depending on the software component responsible for thread management, we can classify threading packages into three traditional models:

[2]POSIX refers to Portable Operating System Interface. It defines a standard operating system interface and environment.

1. **User-level**: Threads created within a process are invisible to the kernel scheduler. This means that user-level threads have to be scheduled by a runtime system which is part of the process or part of the threading package. Therefore, the operating system is not involved at all regarding context switch of threads, except the main process. While user-level threads offer better performance and flexibility due to the low overhead incurred by context switches (less implication of the OS), user-threads have a problem: a thread making a system call will block the whole process with all its threads until the system call returns. Moreover, since user-level threads are invisible to the OS, one thread only within the process has the possibility to take the CPU resource, limiting the ability of the application to use multiple processors (cores) offered by the platform. An example of this threading model is the `POSIX/ANSI-C` based `GNU Pth (Portable Threads)` library [GNU] for `UNIX`-like platforms.

2. **Kernel-level**: Unlike user-level threads, kernel-level threads created within a process are visible by the operating system. Hence, context switch and scheduling of threads is fully performed at the kernel level. Kernel-level threads do not have the problem of blocking when making system calls: a system call blocks only the calling thread. Another benefit, kernel threads take better advantage of multiple processors, hence increasing the amount of parallelism that can be achieved. However, switching among threads can be more time-consuming due to the OS implication. The current implementation of threads in the `Linux` kernel is the `NPTL (Native POSIX Threads Library)` [DM03].

3. **Hybrid-level**: As seen above, user-level threads offer better flexibility and performance than kernel threads. On the other hand, kernel threads do not have any problems with I/O blocking problems and take better benefit from multiple CPUs. So the idea behind a hybrid model is to combine the advantages of both models, while avoiding their disadvantages. A hybrid library creates a number of kernel threads, capable to execute a number of user-level threads. We talk about an `N:M` model, where N user-level threads map onto M kernel threads. This is a compromise between kernel-level `1:1` and user-level `N:1`. Solaris offers this kind of model [PKB+91].

TBB (Intel Threading Building Blocks)
TBB [Int] is a generic library that extends the ISO C++ language for an efficient use of multicore architectures. Like OpenMP (to be presented in Section 2.2.2.2), TBB is designed to promote scalable data parallel programming. To use the TBB library, the programmer has to specify TBB tasks instead of threads. Tasks represent portions of code that might run independently and concurrently. The library itself maps these tasks onto physical threads and processors for an efficient cache utilisation and load balancing. A specified concurrent task is split into independent tasks by the library, each task processes a subset of the data. This approach allows to leverage the power of multicore processors, while ignoring issues of parallelism such as low-level threading constructs or the scheduling and distribution of computations. TBB employs generic programming or template-based programming in C++; many of the library interfaces are defined by requirements on data types and not on specific types, thus, allowing to write flexible and efficient code.

2.2.2.2 Parallel programming languages for shared memory machines

This section discusses Cilk and OpenMP.

Cilk
Cilk [FLR98] is an extension to the C language and a runtime for efficient multi-threaded programming on shared memory multiprocessors developed at MIT. The Cilk language consists of C with the addition of some keywords to indicate parallelism and synchronisations. Consequently, it is the responsibility of the programmer to structure his code in order to expose parallelism. By doing so, the programmer lets the Cilk runtime deal with the low-level thread management and computation scheduling in order to run efficiently on a given platform. Selected sections of code that can run in parallel are translated to Cilk threads which are mapped onto physical threads by a Cilk runtime. The execution model of Cilk is the following: a set of physical threads executing Cilk threads are created at the beginning of the application, the number of physical threads corresponds to the number of available processors. The Cilk runtime implements a *Work-Stealing Scheduler*; each physical thread maintains a work queue of ready Cilk threads and manipulates the bottom of the queue like a stack. When a physical thread's queue becomes empty, it steals a Cilk thread from a randomly selected queue of an another thread.

Open Multi-Processing (OpenMP)
OpenMP [Theb] is a specification for a portable Application Program Interface (API), it aims to facilitate parallel programming of a range of shared-memory parallel machines. It defines a collection of compiler directives, library routines and environment variables to parallelise sequential programs written in both Fortran and C/C++ languages. Like MPI, OpenMP is a joint collaboration between major hardware and software vendors. The goal is to provide a portable model for developing parallel applications across a variety of computer architectures, operating systems and compilers. The compiler directives allow to extend the C/C++ and Fortran languages with loop-based parallelisation, tasking, work-sharing and synchronisations constructs. What makes the OpenMP model more attractive from the programming perspective, is the reasonable effort required to achieve the parallelisation of sequential programs. Indeed, by using compiler directives, the programmer has only to specify which sections of code to execute in parallel or to specify potential parallel loops associated to some policies for iterations distribution. After compilation, the compiler generates code for automatically creating a set of threads (equal to the number of available CPUs/cores seen by the OS by default), distributes iterations among threads by applying the user-specified policy or a static default one, synchronise threads at the end of the parallel region and finally terminates the execution of threads. It is also possible to use constructs providing support for sharing and privatising data.

Using TBB or OpenMP depends on the code structure and the developer objective. OpenMP is much more easier and offers an incremental way to parallelise code, it keeps the code clean and easier to maintenance unlike TBB that needs major changes to the code. Moreover, OpenMP is a standard for programing shared memory machines. On the other hand, TBB's advantage is that the programmer does not need to understand how threads work, he has just to specify sections of code that could run concurrently and let the library map the tasks onto threads. Another advantage of using TBB is that it matches well with code that is highly object oriented since it makes heavy use of C++ templates and provides thread-safe and concurrent containers and some generic parallel algorithms.

2.2.3 Virtual shared memory programming

Virtual shared memory model is also known as distributed shared memory or partitioned global address space model. It is a high-level abstraction of a distributed memory machine (or a set

of interconnected machines as well) that allows the programmer to see the machine as a global shared addressing space. This abstraction can be provided by either an operating system, a library or a compiler. The goal is to hide to the programmer the details of low-level communications primitives. The processes of an application can access local (private) data or distributed shared data. The runtime ensures synchronisation and coherence. The advantage of virtual shared memory programming is to free the programmer from explicit communications, and consequently to use shared memory programming model. However, this comes at the expense of performance loss due to the high overhead incurred by the employed memory coherency mechanisms. There exists multiple language extensions to support the virtual shared memory programming model. We can think to Co-array Fortran [NR98] which consists of Fortran 95 extensions, the Titanium language [YSP+98] which consists of Java language extensions to support parallel computing, the Chapel parallel programming language [CCZ07] or the object-oriented X10 programming language [CGS+05]. The later two languages are designed as new languages rather than language extensions.

We present in the next paragraph UPC which is another language belonging to the virtual shared memory programming model.

UPC (Unified Parallel C)
UPC [CDC+99] is an extension to ANSI-C programming language for parallel processing. UPC follows the SPMD parallel programming model and supports a partitioned global address space. In other words, it supports a distributed/virtual shared memory model. In UPC, the programmer has explicit control over data distribution across threads. UPC offers constructs that allow the programmer to declare data as shared or private to each thread. It aims to exploit memory locality by placing data as close as possible to the threads that use that data. On one hand, private data of a given thread can not be accessed by other threads. On the other hand, shared data are logically partitioned into fragments, where each thread owns a private fragment. That is, each thread has a logical association or affinity to the portion of data that is assigned to it. However, independently of fragment association, all threads can access the whole shared memory space. Besides data distribution constructs, the UPC memory model offers also memory consistency constructs to ensure coherency of the declared shared data. With the UPC memory model, shared accesses are either strict or relaxed. Strict memory accesses issued by a given thread always appear to all threads as being executed in a sequential program order. Relaxed shared memory accesses issued by a given thread may be reordered by the implementation. In this case, other threads may not see these accesses as in a sequential program order.

Applications written using UPC may have poor program performance if data locality is not ensured regarding the computations performed by the intervening threads. In fact, if all threads make heavy use of large portions of shared data, the risk of false sharing is high. Moreover, ensuring memory consistency may require a heavy use of data synchronisation constructs. Consequently, it may degrade performance in a large extent.

2.2.4 Hybrid programming model

Hybrid models combine multiple parallel programming models in the same program. The strength of such models is to take benefit from each composing parallel programing model, while limiting their disadvantages, thus adapting for particular situations. For example, using

MPI with OpenMP can be considered as a hybrid programming model. It is then possible to use shared memory programming with OpenMP inside a NUMA node and message passing with MPI for inter-node communications. As an implementation example of this programming model, we present in the next section a framework called MPC.

MPC
MPC is unified parallel framework for HPC clusters of NUMA machines [PJN08]. Its main goal is to unify multiple parallel programming models inside a single framework in order to efficiently exploit parallel machines. Mainly, MPC is a thread library. It proposes three programming models:

1. Shared memory programming:
   (a) POSIX thread;
   (b) Intel TBB;
   (c) OpenMP
2. Message passing programming:
   (a) MPI;
3. Hyerid MPI/OpenMP

MPC implements a non-preemptive user-level threads package which is compatible with POSIX-threads. In this case, threads created in MPC are not visible by the operating system kernel. That is, an MPC thread manager does all the scheduling management. In addition, by recompiling the open source version of TBB, MPC is able to run TBB applications. Moreover, MPC proposes OpenMP and thread-based MPI implementations which are highly integrated (using MPI/OpenMP). Regarding the MPI implementation, instead of using processes, MPC uses a user-level thread for each MPI task.

2.3 Conclusion of the chapter

With the rise of parallel machines that support the simultaneous execution of multiple threads in parallel, has come the need of languages, compilers, runtimes and operating systems that support and exploit these resources. Several software studies were made in this direction in order to take benefit from these machines and many programming models and implementations were proposed. However, the rising complexity of the hardware makes it difficult to generalise one solution among all kind of machines. Moreover, the intervention of the programmer is increasingly required to achieve decent program performance. Indeed, improving the performance requires the knowledge of internal characteristics of the hardware, both micro-architectural and architectural. Applications have to be written with an explicit parallelism to be able to exploit all the available resources. Of course, there is a trade off between the programming effort and the expected performance improvement.

Nowadays, multicore platforms are everywhere, from the cheapest embedded system to the very expensive supercomputer. Getting the best performance from these machines is even more complicated and crucial. There are hundred of parallel languages, runtimes and operating systems that worked very well for past multiprocessors and distributed machines but, most often, they are not anymore adapted for current hardware designs. So applications written on top

of these software technologies have to be re-written or adapted to tackle these new designs and constraints. For instance, taking into account the hierarchy of shared caches present in multicore processors is mandatory if the programmer wants to get good performance from a given architecture. Indeed, let us consider the cases of OpenMP and MPI for instance. The former proposes a flat memory model where all threads are supposed to access memory with a similar behaviour (uniform latency of access to memory). The later proposes a flat communication model where the communication latency is considered to be uniform between all processes. From the programming perspective, these abstract models of the machine simplify programming. However, from the performance perspective, these flat models are not adequate since they do not account for memory hierarchy (multiple cache levels) or for NUMA effects that make the threads/processes do not access memory or communicate in an homogeneous manner.

Multiple efforts have been done to enhance the expressiveness of programming models in order to efficiently exploiting heterogeneous machines. The OpenMP standard (version 3.0) was extended by the concept of tasks and by nested parallelism. OpenMP tasks are generated inside a parallel region, then each task can be executed by the threads of that parallel region. Nested parallelism allows to nest parallel regions inside others allowing a hierarchy of parallelism. Similarly, the OpenMP execution runtime developed in [oBFG+10] exploits the concept of a hierarchical parallelism. A topology-aware OpenMP thread scheduler is implemented allowing to distribute threads into groups and to map each group of threads in a certain NUMA node or socket for instance. Heterogeneous machine designs can also consist of the combination of CPUs and accelerators like GPUs (Graphical Processing Unit). In this context, we can consider the case of OpenCL (Open Computing Language). OpenCL is an open standard for parallel programming of heterogeneous systems, it aims to exploit the power of state of art multicore processors and GPUs using user specified tasks or kernels.

In this thesis we study the performance of applications following the shared memory programming model (OpenMP and Pthreads) and running on shared memory multicore machines. As we noticed earlier, these programming models offer a flat memory model. This means that while from the application programming perspective, memory accesses latency is considered as uniform (same access latency whatever the location of the data), it is actually not true from the performance tuning perspective since we are dealing with hierarchical designs. The next chapter gives an overview on some performance optimisation issues when it comes to run multi-threaded applications on multicore architectures.

Chapter 3

Related Work on Multicore Performance Evaluation and Tunning

This chapter presents an overview on performance evaluation methodologies. Besides, it presents software and hardware techniques to improve shared cache performance in multicore architectures. It is organised as follows. In our study, depending on the experimental setup, we observed non-negligible performance variability of some high performance codes. It is clear that if this instability is not rigorously considered, the conclusions of the performance evaluation and measurement may be misleading. In this context, Section 3.1 discusses the problem of program execution times variability and introduces a statistically rigorous performance evaluation protocol. In our effort to quantify and qualify the various factors that can influence the variability of program execution times, we found that thread affinity plays an important role. However, it is not clear how thread affinity contributes to increase or decrease the performance variability. In order to understand these interactions, we studied two aspects: 1) data sharing/reuse and 2) shared cache contention. These concepts are introduced in Sections 3.2 and 3.3. The former (Section 3.2) presents software techniques to measure data locality of single-threaded and multi-threaded applications. The later (Section 3.3) discusses the impact of cache sharing on the overall performance of co-running workloads. Finally, in multicore architectures with a hierarchy of shared caches, running a multi-threaded application can yield multiple thread affinity placements, where each can have a distinct performance impact regarding cache performance. In order to compute effective strategies as far as data reuse is concerned, Section 3.4 describes techniques to exploit data sharing by thread affinity in multicore processors.

3.1 Variability of program execution times

Performance analysts often consider the program execution time as the first metric to investigate in the process of performance evaluation, for instance comparing the execution times of two versions of the same program compiled with two different compiler optimisation flags. Unfortunately, performance data can be polluted by errors or noise that can affect experimental results. These errors or noise can come either from the experimental environment of the experiment: hardware and/or software or from the measurement itself (the act of measuring perturbs the program being measured). For example, there is a time required to read a timer before the code to measure and store the timer after this code. Thus, if we execute a program N times, we may obtain N execution times. The phenomena of observing these N distinct execution times

is called *variability of program execution times*. Program performance optimisations, feedback-directed iterative compilation and auto-tuning systems [KSP09] all assume a fixed estimation of execution time given a fixed input data for the program. However, in practice we observe non-negligible program performance variations on hardware platforms. These variations appear differently, depending on the applications and platforms.

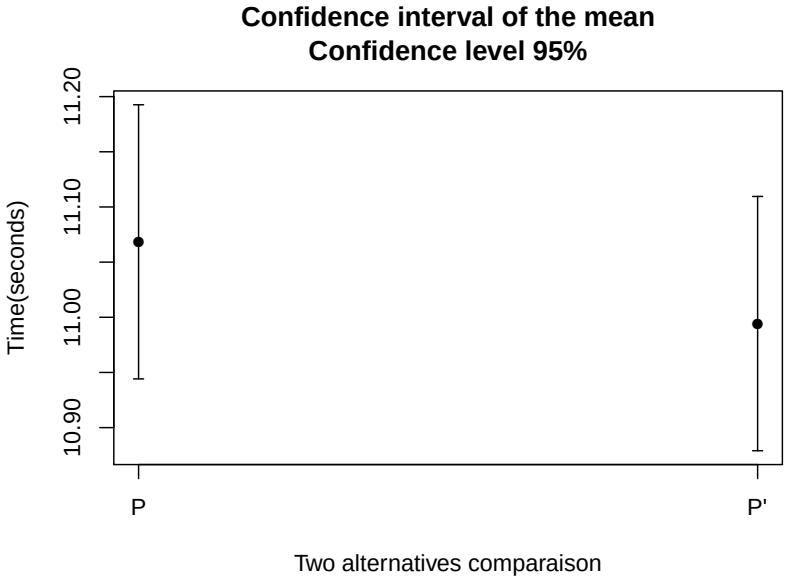


Figure 3.1: Comparing between two confidence intervals of the mean

When a program exhibits a large variability of program execution times, the conclusions about the true performance behaviour of the program are hard to derive. Indeed, the main problem is what is the real performance of the program? In this context, if we do not consider a fixed estimation of program execution times, then we have two answer two questions: 1) what kind of metric should be used to summarise the performance of the program? And 2) how to compare the performance of multiple configurations of the same program? To illustrate this situation, suppose that we have two samples of program execution times X and Y[1]. While X represents execution times of a program P, Y represents execution times of a program P' after applying an optimisation technique to P. In order to compare between P and P', we report in Figure 3.1 the average execution times and the confidence interval (CI) of the average with confidence level of 95%. If we consider only the average execution time, we may conclude that P' is better than P (the optimisation works). However, due to variations of execution times, the CIs of the average execution time of the two configurations overlap and the average value of Y is in the CI of X. This means that this time, we can not conclude that P' is better than P [Raj91].

Even if CIs do not overlap, it does not mean that really the two alternatives are different. Consider again the same example presented above with the exception that X and Y have different data. Figure 3.2 reports the execution times using the average and the CI of the average execution time. Since the CIs do not overlap, we can conclude that X is higher than Y. (P' is better than P). However, if we consider the median execution time (diamond point for P and triangle point for P'), it is possible to conclude that there is no difference between the two

[1] X and Y are real performance data measured in real experiments

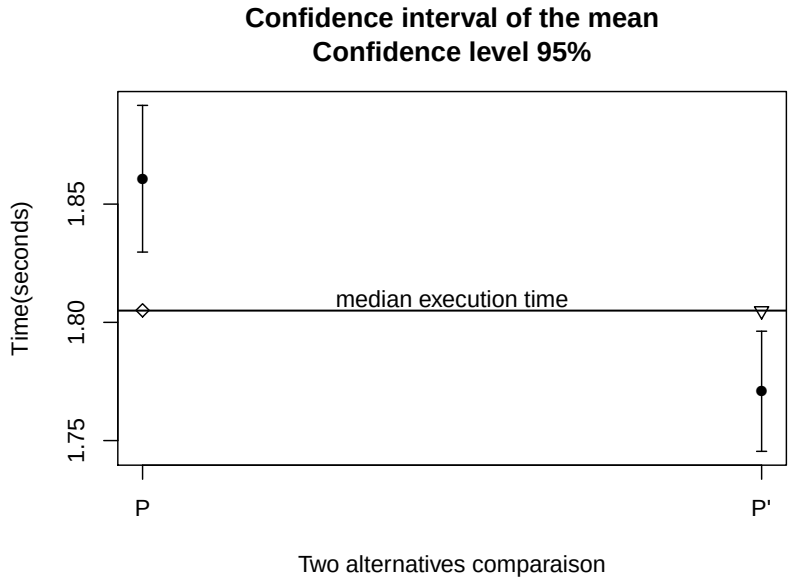


Figure 3.2: Comparing between two confidence intervals of the mean

configurations.

To deal with performance variations, we need two use three complementary approaches:

1. Given an evaluation environment, identify the factors that influence the most on performance variations. This aspect is one of the goals of this thesis, it is presented in Chapter 4. Some of these factors are presented in Section 3.1.1.

2. Better control on the experimental setup and use of a rigorous performance evaluation methodology. When a performance evaluation is performed, the experimental setup (whether hardware or software) has to be fixed in order to reduce as much as possible the variations and reproduce results (this thesis presents an example of such methodologies).

3. Use statistical analysis protocols for comparing the program performance of multiple versions. Section 3.1.3 discusses some examples of such protocols.

The next section presents some factors that may influence on program execution times variability.

3.1.1 Factors influencing the variability of program execution times

There are multiple factors that can make program execution times to vary. We classify these factors into four distinct classes:

1. Inherent to measuring errors.
2. Inherent to the program:
   (a) synchronisation and lock contention;
   (b) OS calls;
3. Inherent to the execution environment:
   (a) Machine workload;

   (b) Starting stack address [MDHS09];
   (c) Thread and process placement;
   (d) Variable CPU frequency;
4. Inherent to the micro-architecture:
   (a) Memory hierarchy;
   (b) Out of order and speculative execution;
   (c) Data prefetching;
   (d) Branch prediction;

The next section gives some related work on quantification and qualification of variability of program execution times.

3.1.2 Quantifying and qualifying variability of program execution times

We can classify performance variability studies mainly in two classes: 1) variability of program execution times or 2) variability of hardware performance counters. In the former class, we can consider the variability of program execution times in native executions whether for the whole program [MDHS09, Gri09], for program fractions [HKA+01, Hug09] or for simulated programs [AW03]. Regarding hardware counters, it possible to consider the variability of the number of executed instructions [WM08] or comparative studies between multiple hardware performance counters access infrastructures [ZJH09].

Hughes *et al.* [HKA+01] studied the variability of program execution times of multimedia applications running on top of general purpose processors. They focused on the analysis of the *frame-level* execution time variability. The analysed multimedia applications periodically process a set of data, the processing of each piece of data is commonly called a frame; each frame has a constraint that it must be completed in a certain deadline. Using simulations and some real machine measurements, they observed that most of the analysed multimedia applications exhibit frame-level execution time variability. They observed that the *frame-level* execution time variability is in the range of 37% and 195%. They concluded that the variability is mostly caused by the application algorithm and the media input rather than the architecture. They considered that if there are variations in the instruction counts, then this variability is inherent to the application (standard deviation is up to 27% in some cases). On the other hand, variations in the IPC (instruction per cycle) means that the variability is due to the architecture (standard deviation less than 5%). They also concluded that aggressive architectural features induce little additional variability and unpredictability.

Collective optimisation [Gri09] is a valuable effort in the community of program optimisation aiming to log performance numbers in a central database. One of the main motivations behind this effort is the disparity of performance scores reported in the literature, and the difficulty in comparing, checking and reproducing them. A fraction of the non reproducibility of experimental code optimisation results comes from the variability of program execution times; if not correctly reported or evaluated, the overall reported speedups would have a low chance of being reproduced.

Another effort dealing with variability is the work of Leather *et al.* [Hug09]. They proposed a performance optimisation system based on observing the execution time of code fractions (functions and so on). The average execution time of such code fraction is analysed thanks to

the Student's t-test, aiming to compute a confidence interval for the average execution time.

Alameldeen *et al.* [AW03] studied time and space variability in architectural simulation studies of multi-threaded workloads (transactional workloads). Time variability occurs when a workload exhibits different characteristics during different phases of a single run, it means that the execution time for each processed transaction may not be similar. Space variability occurs when two runs exhibit different execution times. For instance, in our work we focus on the later definition of performance variability. Regarding time variability, they observed variations up to 31%. In the case of space variability, they found that for the tested benchmarks, the variability exceeds 3% in almost all the tested benchmarks. Interestingly, they also observed that variability decreases when the applications run for longer periods of time. They conclude that ignoring these variations can lead to incorrect results when comparing architectural designs using simulations.

Variability of program execution times has been shown to lead to wrong conclusions if some execution environment parameters are not kept under control [MDHS09].

For instance, the experiments on sequential applications reported in [MDHS09] show that the size of Unix shell variables may influence the execution times. In fact, the size of Unix shell variables affects the starting address of the stack which in turn affects memory alignment (up to 5% variations in execution times). They also showed that the linking order of object codes may influence the execution times. Regarding these two parameters, a performance analysis may overestimate/underestimate the performance of an application or lead to incorrect conclusions.

Contrary to the variability of execution times, Weaver *et al.* [WM08] studied the variability of instruction counts across multiple runs (7 runs) and multiple processor platforms (processors from Intel and AMD, benchmarks from SPEC CPU2006 and SPEC2000). The number of executed instructions is measured in two ways: 1) using hardware performance counters (the retired instructions counter measured with the `perfmon` [Era04] tool) and 2) using dynamic binary instrumentation (using the `Pin` [LCM+05] and `Valgrind` [NS07] frameworks).

They considered multiple sources of variations:

- The accuracy of the counter. Each platform provides a specific counter for retired instructions. However, what the counter really does, differs from one platform to another. For instance, the instruction `fldcw` is counted as two retired instructions in some processors whereas it is counted only once on others.
- The virtual memory layout. Some applications are sensitive to virtual memory layout, for instance the size of environment variables [MDHS09] can lead to significant performance variability.
- System effects: page faults, I/O and number of timer interrupts.
- Variability incurred by dynamic binary instrumentation tools.

The measurement study compares two configurations: 1) a naive execution of CPU2000 and CPU2006 (the experimental setup is not fixed), and 2) a more careful run by following a fixed measurement methodology (we discuss a similar methodology in Chapter 4).

To quantify variability, they used the coefficient of variation (CoV) metric. The CoV is defined as the standard deviation divided by the average. Using an overall estimation of the variations of benchmarks across all the machines, they conclude that with a naive execution, the coefficient of variation of instruction counter is up to 1.07%. They observed that after

fixing the experimental setup, the variations are less than 0.002% for all the benchmarks. They found most of the variability of instruction counts is due to the virtual memory layout (some benchmarks are sensitive to starting address of the heap and the stack), and to some extent the variations are due to timer interrupts (number of time the timer interrupt is triggered).

In the field of hardware performance counters again, Zaparaunuks *et al.* [ZJH09] studied the accuracy of performance counter measurement tools. They performed a comparative study of three well known measurement infrastructures on three different processors. The tested infrastructures are: 1) `perfctr` [Per], 2) `perfmon` [Era04] and 3) `PAPI` [BDG+00]. They concluded that multiple factors have a non negligible effect on the accuracy of the measurement like: 1) the number of measured performance counters (the number of hardware registers used simultaneously), 2) high level vs low level APIs, 3) kernel mode vs user mode measurement and 4) the duration of the measurement . As an overall estimation for example, they found that when measuring the number of instructions at user level, the measurement error can lead to 2500 user-mode instructions across the tested tools. When counting at user and system level, the error between the used measurement tools can lead 10000 instructions. One may say that 10000 instructions error is not important compared to the total number of instructions of an application (possibly billions of instructions). However, if the measurement is intended for some regions of the code or for program phase characterisation for instance, the impact of misleadings conclusions may be high. Similarly, Moore [Moo02] discussed accuracy and efficiency issues when using `PAPI` with the counting and sampling modes of hardware performance counters.

In the light of the different studies that we presented in this section, we can conclude that variability is commonplace and can not be considered as negligible whether variability of program execution times or instruction counts. We presented some related work discussing possible factors that may produce these variations. Most often, we can notice that non-fixed parameters in the experimental setup is the key factor that leads the execution times to vary [MDHS09]. So, to avoid some of these variations, the experimental setup (whether hardware or software) has to be kept under control. However, this is not sufficient to eliminate all the variations. Still, it is possible to use statistical analysis to limit the influence of outlier (the minimal and the maximal) values when comparing different alternatives. Using statistical analysis allows to be careful regarding the conclusions about the performance behaviour of the applications under study. The next section presents some statistical performance data analysis protocols.

3.1.3 Statistical performance evaluation

In this section, we present two statistical performance evaluation protocols: 1) JavaStats and the 2) Speedup-Test protocols. We present in the next section the JavaStats protocol.

3.1.3.1 JavaSats

Georges *et al.* [GBE07] studied variability of program execution times for `Java` programs. They first showed that Java programs are experiencing non-determinism or variability of execution times. Second, they presented some prevalent `Java` performance evaluation methodologies. These methodologies differ from each other in different ways: 1) measurement methodology and 2) data analysis. In the measurement methodology, they discussed three approaches: 1) iterate the benchmarks multiple times within a single virtual machine (VM) invocation; 2) multiple VM invocations and iterate a single benchmark execution; and 3) multiple VM invocations and iterate the benchmark multiple times. Regarding data analysis, the authors discussed

methodologies that differ in the way they report performance numbers from a given sample of execution times: average or median vs best vs the worst execution time. To overcome the weakness of the previous methodologies to account for performance variability, they advocated the use of a rigorous statistical methodology to compare Java performance. For a single Java program with a fixed input data running on a single virtual machine, they considered performance evaluation methods for two cases:

- Measure startup performance (the performance of the virtual machine):
  1. Take multiple measurements, each comprises one VM invocation and a single benchmark iteration;
  2. Compute confidence intervals for the average execution time across these measurements.
- Measure steady performance (the performance of the program itself)
  1. Consider p VM invocations and q benchmark iterations *i.e.* we have $p \times q$ measurements;
  2. For each $1 \leq i \leq p$ invocation, retain only k measurements from the q iterations. The k^{th} iteration is reached once the coefficient of variation (CoV) of these k iterations falls below a fixed threshold;
  3. For each VM invocation compute the sample mean of the retained k measurements;
  4. Compute the confidence interval across the computed means from multiple VM invocations

After computing confidence intervals for the average execution time, the authors consider two cases: 1) comparing two alternatives and 2) comparing more than two alternatives. The former computes a confidence interval for the difference in the two samples average. If the confidence interval includes zero, there is no statistically significant difference between the alternatives for the chosen confidence level. If the confidence interval does not include zero, the sign of the average difference indicates which alternative is better [Raj91]. The later considers an analysis of variance (ANOVA). An ANOVA analysis tests if there is a statistically significant difference between all the alternatives. To know between which alternatives, there is or there is not a statistically significant difference, the authors use the `Tukey HSD` (Honestly Significantly Different) test. The `Tukey HSD` test allows to compare all the possible pairwise averages of the tested alternatives with a risk level α and find which average value is significantly different from another.

3.1.3.2 The Speedup-Test protocol

The statistical protocol for performance analysis proposed by [GBE07] focused on average execution times only. It is well known that the average is sensitive to outliers (minimal and maximal values); so relying heavily on it may not be appropriate. For that reason, the median is usually advised for reporting performance numbers. Touati *et al.* [TWB12] proposed a rigorous statistical methodology (The Speedup-Test protocol[2]) based on well-known statistical tests to study the statistically significance of observed speedups. By fixing a confidence level α, the protocol is able to compare between two sample averages or two sample medians.

The Speedup-Test uses the following tests:

[2]The Speedup-Test protocol is implemented and distributed as an open source tool based on the `R` software. In the remainder of this document, we rely on this protocol to certify the significance of our computed speedups

1. Shapiro-wilk test. It checks for the normality of a sample[3] of program execution times X. In other words, it checks if the sample X follows the normal distribution.

2. Fisher F-test. For a risk level α, it checks if two samples of program execution times X and Y have similar variances $\sigma_X^2 = \sigma_Y^2$.

3. Student's t-test. For two samples of program execution times X and Y, the student's t-test checks with a risk level α for the null hypothesis $\mu_X \leq \mu_Y$ where μ_X and μ_Y are the average values for the X and Y samples respectively.

4. Welch's test. This test is an adaptation of the Student's t-test. It is used when two samples of program execution times X and Y have unequal variances.

5. Kolmogorov-Simirnov test. It checks if two samples of program execution times X and Y follow the same distribution.

6. Wilcoxon-Mann-Whitney's test. For two samples of program execution times X and Y, the Wilcoxon-Mann-Whitney test checks if the median of X is greater than the median of Y and if $\mathbb{P}[X > Y] > \frac{1}{2}$, which means the probability that an individual execution Y is faster than an individual execution X.

Having two samples X and Y of program execution times, computing a statistically significant speedup of the observed average execution time using the Speedup-Test methodology follows the following protocol:

1. If the two samples are large enough ($|X| > 30$ and $|Y| > 30$), use the Student's t-test with a fixed risk level α[4]

2. If one of the samples is small ($|X| \leq 30$ or $|Y| \leq 30$)

   (a) If X or Y does not follow Gaussian distributions (using the Shapiro-Wilk test) with a risk level α, then it is not possible to conclude about the statistical significance of the observed speedup of the average execution time. In this case more runs are required to build a large sample.

   (b) If X and Y follow Gaussian distributions (Shapiro-Wilk test) with a risk level α then:

      i. If X and Y have the same variance (using the Fisher F-test) with a risk level α then use the standard Student's t-test.

      ii. If X and Y do not have the same variance (using the Fisher F-test) with a risk level α then use the Welch's version of the Student's t-test.

Similarly, the Speedup-Test computes a statistically significant speedup of the observed median execution time for two samples X and Y. The computed speedup relies on the Wilcoxon-Mann-Whitney test to check if the median execution time has been reduced or not between the two samples. Performing the later test follows the following protocol:

1. Perform the two-sided and unpaired Kolmogorov-Simirnov test with risk level α to check that the two samples are from the same distribution.

2. If X and Y are not from the same distribution, then check if X and Y are large enough:

[3] A sample is a finite set of program execution times.

[4] The Student's t-test makes the assumption that the distribution function of the two samples follow a normal distribution. However, if they are not, then it is admitted (but not proved) that the test stays robust for large samples

(a) If $|X|$ and $|Y|$ are large enough ($|X| > 30$ and $|Y| > 30$), then it is known that it is possible to use the Wilcoxon-Mann-Whitney test for large samples with risk level α but the risk level may not be preserved.

(b) If $|X| \leq 30$ or $|Y| \leq 30$ then it is not possible to use the Wilcoxon-Mann-Whitney test, more runs are required.

In addition of computing a statistically significant speedups for the average or median execution times for a given benchmark and a fixed data input, the Speedup-Test protocol gives the possibility to compute two metrics 1) an *overall speedup* S and 2) an *overall performance gain factor* G across a set of benchmarks. The idea behind these metrics is when we apply a code optimisation technique on a set of benchmarks, practically only a fraction of programs will take benefit from it. In [TWB10], they suggested that it is possible to compute the S and G only for the fraction of benchmarks that succeed with the Student's t-test or Wilcoxon-Mann-Whitney. If we consider that we observe a speedup in p out of n benchmarks, then the fraction $\frac{p}{n}$ represents the proportion of accelerated benchmarks. To evaluate if the code optimisation technique is beneficial for a large faction of programs, the Speedup-Test protocol computes a confidence interval with a fixed confidence level α for this proportion of accelerated benchmarks.

3.1.4 Discussion on variability of program execution times

In native program execution, the instability of program performance makes an accurate quantification of program performance highly challenging. Indeed, this variability may lead to wrong conclusions about the true performance behaviour of programs. For example, varying the size of the UNIX shell environment may lead to up to 5% variability in execution times [MDHS09]. Measurement infrastructures like software to access hardware performance counters can also introduce variations in the reported counters, mainly it is dependent to the way that these counters are used [ZJH09]. One would think that using simulators will help to reduce this variability. Unfortunately, not only they are slow, they also present a bias in measurement, more than 3% variability in execution times [AW03]. The sources of the measurement bias are numerous. However, one of the important factors is incorrectly fixed parameters in the experimental setup [MDHS09, WM08]. Still, operating system (OS) effects like process scheduling, page faults management, OS interrupt handlers have also to be considered.

In order to reduce the effects of these variations, and consequently to decrease the probability to be wrong about the performance of a given program or a system, two aspects have to be considered: 1) rigorous performance evaluation methodologies, and 2) use rigorous statistical methods for performance analysis [TWB10, GBE07]. With the former, we think to a fixed software and hardware setups, multiple runs (for a statistical significance analysis), longer runs (to overcome the overhead of the measurement itself) or by using a large number of benchmarks (to overcome the variations that come from the application itself). The later aspect is related to the use of statistics. Indeed, a statistical data analysis does not remove the variations, but it helps to better interpret performance data with fixed confidence levels.

When it comes to program execution time measurement, it is possible to consider two dimensions: 1) the measurement granularity and 2) the program data input. First, the measurement granularity consists of whether we are measuring a fraction of the program or the whole program (*i.e.* single loop, function or the whole program). For small granularity measurement targets, the sensitivity to errors or noise can be significant, and consequently leading to important variations. On the other hand, long running programs may be less sensitive to variations [AW03]. However, in all cases, the size of the sample has to sufficiently large (more than 30 runs [Raj91])

even if it is time consuming for large programs. In fact, multiple statistical tests require a large sample of data [TWB10, Raj91]. As a complementary aspect to the measurement granularity, we can consider how the repetitive runs are performed: 1) a single run where the repetitions are performed inside the program or 2) multiple invocations of the program. When we perform multiple invocations of the program, even if is not always true, the successive executions can be considered as independent. Unfortunately, this situation is not true when we do repetitions inside a single invocation of the program. There are two main reasons for that. The OS does not behave as if the program is launched multiple times from the shell. For example, the startup time of the program is not accounted. The second reason is caches may be warmed by the repetitive executions. Regarding the program data input, the variability of execution times in this case cannot be analysed with the Student t-test or the Wilcoxon-Mann-Whitney test. Simply because when data input varies, the execution time varies inherently based on the algorithmic complexity, and not on the structural hazard. In other words, observing distinct execution times when varying data input cannot be considered as hazard, but as an inherent reaction of the program under analysis. However, an analysis of the variance (ANOVA) [Raj91] may be used for this aspect. In the remainder of this thesis, we focus on the variability of program execution times with a fixed data input.

The presence of multiple levels of memory cache in mulicore processors increases the need for a better understanding of the locality of data and its measurement in termes of performance stability and performance improvement. In this context, the next section discusses metrics and methods to measure data locality in single-threaded and multi-threaded applications.

3.2 Data locality and reuse distance analysis

The organisation of modern computers relies on the implementation of a hierarchy of memory systems. This organisation defines multiple levels: the highest level of the memory hierarchy is the processor's registers, the lowest level is the main memory and the intermediate levels are memory caches. While lower levels have bigger storage capacity but slower time accesses, upper levels (*i.e.* registers and caches) have less capacity but have lower latency access (depending of the level in the hierarchy). When a processor issues a memory operation on a data, it first looks for at the highest level, then at the lower levels (until the data is found).

Since memory caches have lower latency and less capacity, to achieve good performance, it is better to keep the most frequently data on caches as long as possible, hence improving the data locality of the program. We can divide data locality in two classes:

- **Temporal locality**: It refers to the reuse of a given piece of data in a relatively short time duration in the future.
- **Spacial locality**: It refers to the use of data elements which are stored in relatively close memory locations.

The next section discusses techniques to measure data locality.

3.2.1 Measuring data locality

Data locality of a program is not easy to measure, there is no a counter or a ratio that gives an intuition about its locality. However, since data locality is a consequence of a good or a poor behaviour of a program as far as memory hierarchy is concerned, it can be approximated

by some metrics like number of cache misses or cache hits. These metrics can be computed differently according to their dependence or not to a certain architectural platform. So, we consider architecture dependent and architecture independent metrics.

The next sections discusses the case of architecture dependent metrics to measure data locality.

3.2.1.1 Architecture-dependent metrics

One way to measure data locality of a program on a given hardware platform, is the use of an expensive cache simulation approach or by using hardware performance counters. It is then possible to estimate metrics such as cache miss ratio or cache hit ratio for instance. This approach can derive valuable information about the locality of the program by characterising the behaviour of a program when running on top of a fixed hardware platform. For example, it is possible to compare the performance of a program before and after applying a data locality optimisation technique. On one hand, a good impact of applying a data locality optimisation technique translates in a reduction in the number of cache misses and consequently in program execution time. On the other hand, a negative impact translates to an increase in the number cache misses and consequently, an increase in program execution time. The drawback of this approach is the following: performance data are collected for a given hardware with fixed cache parameters (fixed cache size, number of sets, number of ways, etc.), they are not portable across various platforms. So, if locality information is required for different architectures, the program has to be run on each of these hardware configurations (whether direct measurement or simulation). Consequently, the overhead of the profiling may not be negligible.

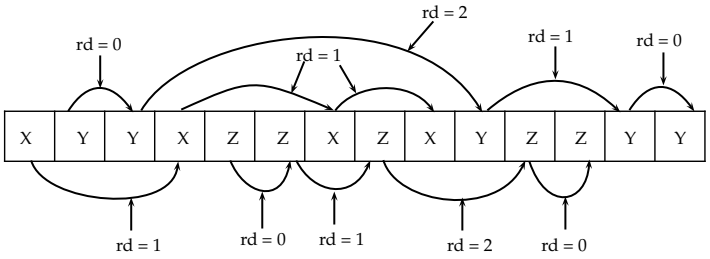
When it comes to cache simulation, we can consider for example the case of `Cachegrind` [Net94]. `Cachegrind` simulates the machine cache hierarchy when a given program executes. It simulates independent first level L1 instruction and data caches and a unified L2 cache. If a machine has also an L3 cache, `Cachegrind` simulates the first level (instruction and data cache) and the last level (L3 cache). `Cachegrind` gathers statistics about hits and misses for each individual source code line. These statistics are related to each tracked level in the cache hierarchy. At the end of the execution, it reports a summary of global statistics for the whole application for each tracked level in the cache hierarchy.

The next section introduces architecture independent metrics to measure data locality.

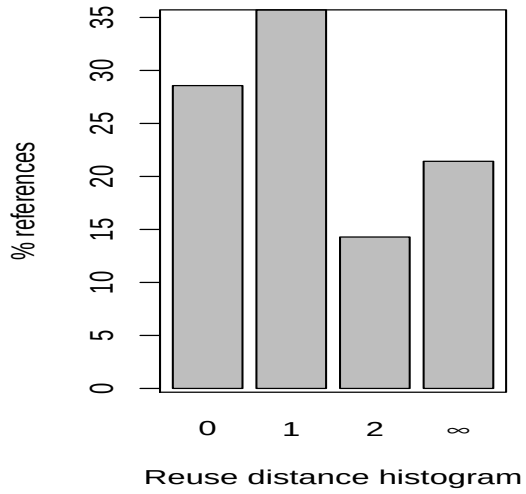
3.2.1.2 Architecture-independent metrics

One of the most influential metrics to measure data locality is the *reuse distance* analysis. Mattison *et al.* studied stack algorithms in cache management and defined the concept of stack distance [MGST70]. Reuse distance is similar to stack distance using the LRU (Least Recently Used) replacement policy. It measures the program locality behaviour of an application in a fully or set-associative cache. It does not depend on any hardware or cache parameters. This allows this technique to measure the locality of programs independently of any particular machine. Most often, reuse distance is used as dynamic technique to approximate the cache behaviour of a program.

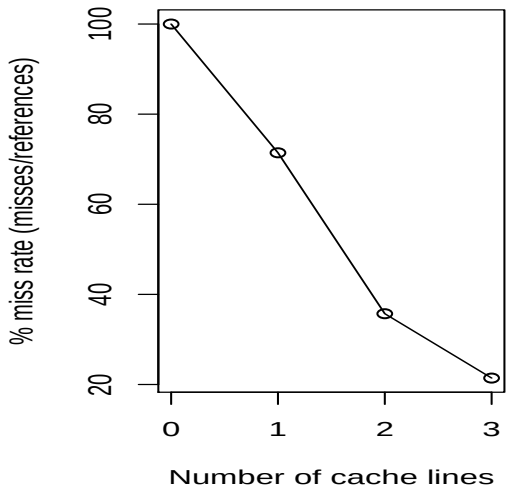
It is also possible to estimate data reuse or data locality at compile time using static analysis. A compiler analyses array references in nest loops of a program, and determines if these references access the same memory locations [WL91, MCT96, Fah97]. It is then possible to classify



(a) A memory references stream



(b) The reuse distance histogram



(c) The miss rate curve

Figure 3.3: An example of reuse distance on a sequence of memory accesses, a reuse distance histogram, and cache miss rate curve for a cache with sizes going from 0 to 3 cache lines

these accesses in terms of data reuse, and by using cost models to predict the cache performance of the program. The idea behind analytical methods for predicting cache behaviour is to model the cache behaviour of a program by means of mathematical formulas. If these formulas can be solved then output of such models like the number of cache misses or the number cache lines a loop nest accesses can be exploited by data locality optimisation techniques. These analytical models include Presburger formulas [CPHL01], probabilistic analytical models [FDZ99] or the Cache Miss Equations [GMM99, VX02]. The main limitation of such analytical models is that they do not account for indirect references, they require known lower and upper bounds of iteration counts, affine expressions or regular accesses patterns.

Due to the limitation of compile-time techniques to accurately model the cache behaviour of programs, we focus our discussion on runtime techniques. The next section presents some related work regarding the concept of single-threaded data reuse distance analysis.

3.2.2 Single-threaded data reuse distance analysis

In a sequential execution, reuse distance is defined as the number of distinct data elements accessed between two consecutive references to the same element [DZ03, DZ01, BD01] or ∞ if

the element has not been referenced before. Figure 3.3a shows an example of reuse distance computation for a memory references stream. We can observe that while the second reference of X has a reuse distance of 1, the second reference of Y has a reuse distance of 0. The granularity of a data element can be a processor word, cache lines, memory page or/and instructions. Moreover, this metric measures the distance (in terms of the number of distinct data accesses) between two accesses to the same data element instead of time.

To measure program locality with reuse distance, a histogram of the reuse of all memory references is used. Shorter reuse distances means good temporal locality because these references are more likely to be cached. On the other hand, longer reuse distances means bad locality because these references are more likely to not be present in the cache when they are referenced again. For a fully associative cache with N lines and with a LRU replacement policy, there are $N+1$ counters: $C_1, C_2, ..., C_N, C_{>N}$. For each cache access, one of these counters is incremented. The $i+1^{th}$ counter is incremented if the data element is found at the i^{th} position on the LRU stack. For instance, if the reuse distance of a data element is equal to zero, then the C_1 counter is incremented by one, if the reuse distance for an access is 1, then the C_2 is incremented accordingly and so on. For all accesses with reuse distances larger than the cache size (the data element is not present in the LRU stack), then the $C_{>N}$ counter is incremented representing the cache miss counter. The $C_{>N}$ counter is also incremented when a memory reference has never been referenced before (first time access equivalent to compulsory misses).

Regarding the above definition of the reuse distance measurement, if we want to study the locality of a full associative cache size having lower cache lines number, says $N' < N$ then the hit counter ($H(N')$) and the miss counter ($M(N')$) can be computed with the following formulas:

$$H(N') = \sum_{i=1}^{N'} C_i \quad and \quad M(N') = \sum_{i=N'+1}^{N} C_i + C_{>N}$$

This formulas is actually a slight adaptation of the formulas presented in [CP03]. The miss rate for a given cache size is computed as the ratio between the number of misses for reuse distances greater than the cache size and the total number of references. Figure 3.3b shows the reuse distance histogram. The histogram reports four bars, each represents reuse distance of 0, 1, 2 and a cold misses counter respectively. A cold miss counter tracks memory locations that are referenced for the first time. Figure 3.3c shows the miss rate curve for a cache with 0, 1, 2 and 3 cache lines size.

Compared to simulating the whole program locality for various cache parameters, reuse distance measurement is faster. However, full reuse distance measurement has the limitation to be slow. The overhead is due to the need to feed the model with all the memory references. Moreover, this overhead is even more prevalent for large data sets. To overcome this limitation, researchers have proposed sampled reuse distance measurement [DZ03, BH04, BH05, SKP10]. The idea behind the concept of a sampled reuse distance analysis is the following: instead of tracking all the memory references, a sampled analysis randomly selects individual references from the dynamic references stream and tracks the selected addresses until their reuse. With this approach, only a subset of memory references have to be tracked to compute the model which can greatly reduce the overhead of the analysis. In this context, statistical models can be used to estimate the miss rate of shared caches. Though fast, the accuracy of a sampling approach can be a problem. Indeed, sampling methods implies to select a sample of random addresses. However, estimating to what extent these selected addresses can be considered as

representative of the full data stream remains difficult.

Reuse distance analysis can also be applied for data locality prediction purpose. It allows to profile few runs of program using different data input. With the help of some statistical models, it tries to predict how the locality behaviour of the program will be affected when running with other (larger) data inputs. In this context, Zhong *et al.* [ZSD09] examined approximate algorithms for measuring reuse distance and prediction methods for modelling whole-program locality.

The next section presents some related work about reuse distance analysis for multi-threaded applications.

3.2.3 Multi-threaded data reuse distance analysis

The previous section presented the single thread data reuse analysis. We showed that it is able to model the program data locality for a large set of possible cache sizes. Unfortunately, this is not sufficient when it comes to predict the locality behaviour of multi-threaded applications running on multicore processors. The problem with the single thread reuse distance analysis is that it is unable to capture the dynamic behaviour of a parallel execution. This behaviour can be translated to the need to understand the following: how the different threads interact as far as the use of a shared cache is concerned? To accurately predict the locality behaviour of a multi-threaded application, a multicore-aware reuse distance analysis also called a *concurrent reuse distance* (CRD) [WY11] has to consider two key parameters:

- **Memory interleaving**. This parameter considers how memory references of the simultaneous execution of different threads interleave. Considering a parallel execution, dynamic parameters (such as scheduling, synchronisation, I/O, etc.) impact memory references interleaving on the shared cache. Hence, the CRD model has to reflect this thread-interleave behaviour to precisely model inter-thread interaction.

- **Data sharing**. Data sharing impacts cache performance into two ways. First, considering the case of data accessed by all threads in read only mode, the first access to the shared data by one thread will effectively prefetch that data to the cache making it available to other threads. The direct consequence of prefetching that data is to avert future cache misses when the data is accessed by other threads. Second, when a shared data is accessed by multiple threads, only one copy of the data needs to be brought to the cache and used by all threads. Moreover, bringing one copy to the cache saves cache space which can be used to hold other memory blocks with positive impact on the multi-threaded performance.

Figures 3.4 and 3.5 show examples on how a CRD analysis has to consider memory references interleaving and data sharing among multiple threads. The first figure shows a concurrent reuse distance profile for two threads sharing data with distinct memory references. It reports the effects of data sharing on the reuse distance profile. The first scenario is related to the reuse distance of the memory reference A. If we consider the case of the first thread, the reference to A by both threads breaks the reuse interval into "$ACBCA$" and "AEA". The new CRD has to consider the RD of 2 (for "$ACBCA$") and RD of 1 (for "AEA") compared to original RD of 2. The second scenario is related to the memory reference C. Referencing C by thread 2 has the effect of prefetching that data to the cache, the direct effect is to avert a cache miss for thread 1. So, instead of having an $RD = \infty$ for the memory reference C by thread 1, the new CRD is equal to 1. The second figure shows the case of uniform and non-uniform interleaving. When

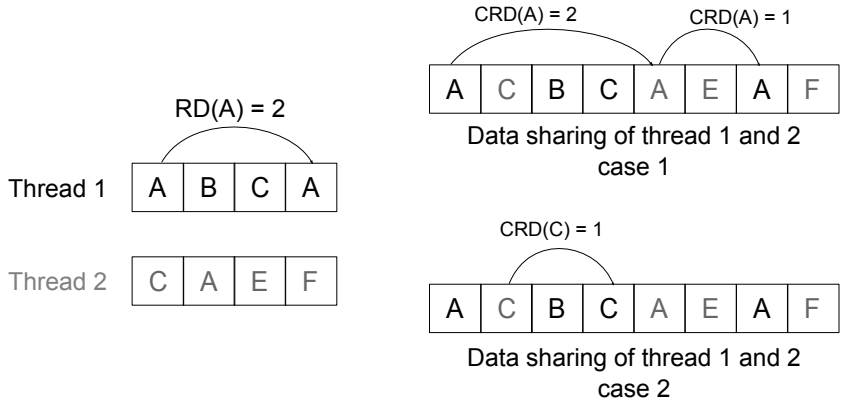


Figure 3.4: A case of a concurrent reuse distance of two threads sharing data

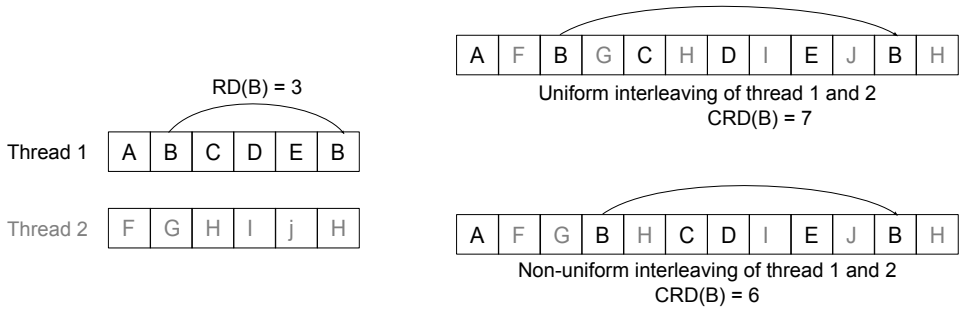


Figure 3.5: A case of uniform and non-uniform interleave of memory references of two threads

it comes to model memory references interleaving, some studies distinguish between a uniform or a non-uniform thread-interleaving (see Figure 3.5). Considering a uniform or a non-uniform thread-interleaved memory references streams depends on how the concurrent reuse distance is computed. Mainly, there are two approaches to build a concurrent reuse distance: 1) one single CRD and 2) merging multiple per-thread reuse distance profiles.

The first approach aims to directly build a global and shared CRD; all memory references of all threads go through this single CRD. In this approach, thread-interleaved memory references is implicit, it is based on the order of arrival of memory references to the shared CRD [SKP10, SPP10, ZKY11]. This approach has one main drawback, it captures one possible memory references order among many other possibilities; thread scheduling and synchronisation for instance, may impact the order in which memory addresses are accessed by threads. Therefore, the reuse distance analysis may vary across multiple profiles.

The second approach [WY11, DC09, JZTS10] aims to build per-thread reuse distance profiles, each per-thread profile captures the classical reuse distance in isolation at the end of the parallel execution. All the individual RDs are merged into a single shared CRD. It is only at the time of merge that the model has to consider the nature of memory references interleave. The merge decision highly depends on the expected behaviour of the multi-threaded application or by answering the question: do threads execute the same code or not? If threads execute the same code, then it is more likely that threads access memory in a similar way, so it possible to consider a uniform model for merging the distinct RDs. On the other hand, when threads execute distinct instruction streams, they are more likely to access memory in different ways. So, it is possible to consider a non-uniform merging model for such applications.

Wu *et al.* [WY11] studied the locality of multi-threaded applications which exhibit loop-level parallelism running on top of tiled multicore processors (using simulators). For these loop-level programs, they made the assumption of uniform memory references interleave and study how CRD profiles scale for larger core numbers. Ding *et al.* [DC09] and *Jiang et al.* [JZTS10] considered the general case of asymmetric behaviour of threads. They considered a non-uniform interleave of memory references to build their CRD models. Although the former approach addresses only one class of multi-threaded applications, it has the advantage of simplifying the problem, since it considers that all threads exhibit symmetrical memory access behaviour. Though general, the later approach employs complex statistical models to account for the highly large number of ways that threads interleave and interfere. For this reason, this approach may be inapplicable for large data sets.

3.2.4 Discussion about data locality measurement

Data locality is the measure of how programs are taking benefit from caching systems. Since the introduction of memory caches, locality has increasingly gained importance in computing systems. Depending on the program locality or the patterns of data reuse and due to the higher speed of memory caches, a memory hierarchy can substantially increase or decrease the program performance [Joh02]. While locality can be measured by hardware-dependent measures such cache miss rates (using hardware performance counters for instance), hardware-independent metrics such as reuse distance, ensures better portability and predictability. Reuse distance can be computed by an exact measurement or by a sampled measurement. Although the exact measurement gives better accuracy, analysing all the memory references is slow. On the other hand, a sampled measurement leads to a substantial overhead decrease but with some accuracy loss. It also possible to measure data locality by static code analysis. This method may be more attractive due to its low overhead (it can be implemented in a source-to-source compiler or a full compiler). Unfortunately, since it is hard to track all the memory references statically, the accuracy of the measurement could be affected (pointer references, data input known at runtime, etc.).

While single thread reuse distance focus mainly on single-threaded programs, it has to be augmented to account for a multi-threaded execution. A multi-threaded aware reuse distance analysis has to consider two aspects: data sharing and memory references interleaving. So, data locality measurement is highly dependent on the two later aspects when it comes to build a concurrent reuse distance. Our work on the data locality techniques presented above differs fundamentally in one aspect. While these techniques focus on how to effectively and accurately measure data locality, this thesis presents runtime techniques aiming to enhance multi-threaded data reuse.

The next section discusses the problem of inter-thread cache contention in multicore processors. This problem happens when multiple independent applications run simultaneously and access to common caches. Indeed, co-locating on the same cache multipe processes may lead to significant performance variability and leads to severe performance degradation on the other.

3.3 Improving shared cache performance for co-running applications

In multicore processors with shared caches, the concurrent execution of multiple processes can result on a destructive interaction leading to a severe performance degradation. An example of such destructive interaction is the case of two programs A and B running on a neighbouring cores accessing to a shared cache. Let us consider the following scenario: while program A exhibits a streaming behaviour on a large working set, program B operates on a small working set, but with good temporal locality. The performance of the program B will be highly affected by the execution of program A. The later will occupy a large footprint on the cache without any temporal reuse. This will lead to evict useful cache lines of program B, hence, degrading its performance.

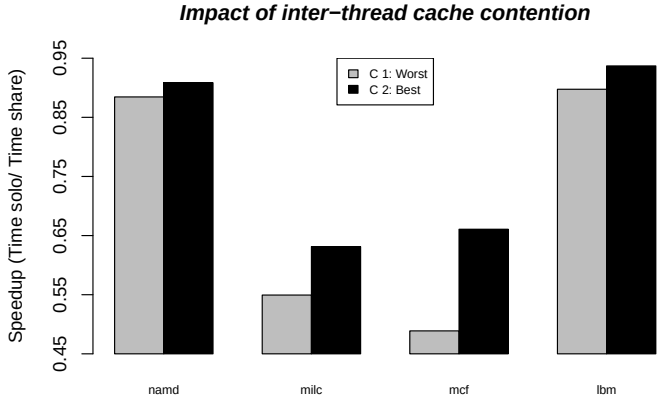


Figure 3.6: The observed performance degradation of four SPEC CPU2006 benchmarks running under three different co-schedules on an Intel quad-core processor (higher bars means low performance degradation)

In order to illustrate the performance behaviour when running multiple concurrent applications on a multicore processor, we followed the idea presented in [ZBF10], and performed an experiment using four applications from SPEC CPU2006 benchmarks[5]: `namd`, `milc`, `mcl` and the `lbm` benchmark. The four applications were run simultaneously on an Intel quad-core processor, where each couple of cores share an L2 cache[6]. Each application runs with one thread and is placed in a distinct core. Each execution of the four applications under multiple thread placement configurations is called a co-schedule.

Figure 3.6 shows the performance degradation of the four applications when running simultaneously under different co-schedule configurations. With four cores and two L2 caches, there are three unique ways to co-schedule the four applications. For each co-schedule, the median execution time (35 runs) of each application is reported. Besides, we measured the execution time of each application when it runs alone. For each application i, and for each co-schedule j, we computed the speedup of running the application i under j relative to a solo run as the

[5]SPEC CPU2006 are single threaded sequential applications.

[6]The machine has two sockets, each with four cores. We use only one socket for our experiments

following: $\frac{Time_i^{solo}}{Time_i^j}$ where $Time_x^y$ represents the reported execution time of application x under co-schedule y. Figure 3.6 reports for each application the best and the worst observed speedups. It shows that depending on applications, the performance degradation due to sharing the L2 cache may be significant.

The impact of cache sharing on program performance in multicore architectures was studied in many research papers, mainly focused in architecture design and in operating system (OS) process scheduling. These studies have different objectives like the increase of system throughput, overall fairness or quality of service. Achieving these objectives requires a better understanding of the interaction between threads in order to: 1) propose prediction models or classification schemes of cache sharing behaviours and 2) propose solutions and techniques that aim to minimise the inter-thread cache misses and contention by placing the different programs appropriately. Most of these studies focused on single threaded applications and target multi-programmed environments.

The next section discusses some related work on inter-thread shared cache contention prediction models and classifications schemes of cache sharing behaviours.

3.3.1 Predicting inter-thread shared caches contention

The first approach to alleviate the problem of cache contention relies on finding a co-scheduling that minimises capacity misses and cache access contention. The advantage of implementing a process scheduling policy is flexibility. The scheduling algorithm can be implemented into two distinct locations:

1. At system level or inside the OS scheduler: this solution is more attractive, since it allows to monitor all the processes running on top of the system.
2. At user level: this solution allows to monitor only a subset of processes. In a high performance system for instance, it is possible to consider only high consuming time applications.

In addition to the implementation location, a software solution for process scheduling can adapt to a dynamic behaviour of applications.

Tackling cache contention purely by process scheduling requires the knowledge of some information or characteristics about the running applications. For that reason, most often these techniques are profile guided. The profile can be collected in two ways:

1. Each application is run once until termination, analyse the profile and apply a scheduling policy.
2. It is not necessary to run the whole application until termination, just take a profile on an sample of the execution and apply the scheduling policy.

Program profiling is important because it allows to build models to understand and predict how applications interact with each other. Despite its advantages, profiling may be highly time consuming. For this reason trade-offs have to be done between accuracy and speed.

Many performance models were studied to predict the impact of cache sharing on co-scheduled applications [CGKS05, XL08, KBH+08, ZBF10]. Chandra *et al.* [CGKS05] studied the impact of L2 cache sharing on threads that simultaneously share the cache on a chip multiprocessor (CMP). They proposed three performance models to predict the impact of inter-thread

cache sharing on the performance of each co-scheduled thread that shares the cache. They observed that cache contention can significantly increase the number of cache misses of a thread in a co-schedule, and showed that the degree of such contention is highly dependent on the thread-mix in the workload. The input of the proposed models is the isolated L2 cache reuse distance of each thread. A reuse distance profile captures the temporal reuse behaviour of an application in a fully or set-associative cache. The output of the models is an estimated number of extra L2 cache misses for each thread due to cache sharing.

In the spirit of predicting the performance degradation experienced by co-running applications when sharing common caches, Zhuravlev *et al.* [ZBF10] investigated contention-aware scheduling techniques to mitigate the contention on shared caches in multicore processors. In order to prototype efficient scheduling techniques, they studied some well-known classification schemes (prediction models) in the research community. The studied models aim to predict the impact of cache sharing on the overall program performance of co-scheduled applications. In other words, the prediction models estimate how likely each thread can affect another when competing for shared caches. After studying the Stack Distance Competition (SDC) [CGKS05], Animal Classes [XL08], Solo Miss Rate [KBH+08] and the Pain Metric (proposed by [ZBF10]) models, they selected the best scheme to design a scheduling algorithm.

A Stack Distance Competition model based on reuse distance analysis. It tries to build a new reuse distance profile that merges individual reuse distance profiles of threads that run together. The output of the model is an estimation of the number of extra cache misses due to the cache competition access.

Animal Classes model classifies cache behaviour of benchmarks into four classes based on simple heuristic metrics. Each class is related to the behaviour of a specific animal with respect to its use of the shared cache. The model uses metrics such the total number accesses to the L2 cache, the total number of L2 misses if the program use the whole n ways of the cache, the relative miss rate if the program has exclusive use of the n ways of the cache (number of misses per accesses) and finally the smallest number of ways needed to achieve a miss rate that is greater than or equal to $k\%$ of solo miss rate. The combination of these metrics allows to define four profiles (animals) of applications:

1. Low-rate access to the shared cache.
2. Frequent L2 accesses but the miss rate is reasonable even if the number of used cache ways is small.
3. Frequent L2 accesses and require an adequate number of ways to achieve good performance (very-sensitive to co-running applications).
4. Frequent L2 cache misses with a high cache miss rate whatever the allocated cache size (frequently hurts other applications).

A Solo Miss Rate model uses hardware performance counters to measure the number of misses or the cache miss rate. This metric can give the scheduler hints about applications with high-rate cache misses. Therefore, it is possible to spread-out these applications onto multiple shared caches in such a way that no cache will experience more cache misses than an another.

The Pain metric model uses the concept of the *cache sensitivity* and *cache intensity*. Sensitivity measures how much an application will suffer when it shares the last level cache. Using probabilities, the model estimates how likely hits in the reuse distance profile turn into misses due to cache contention. The Intensity metric measures how much an application will hurt

other applications. It is measured as the number of last-level cache accesses per one million instructions.

Besides contention on shared caches, Zhuravlev *et al.* [ZBF10] studied other factors causing performance degradation of co-running applications running on chip multiprocessors (CMPs). They provided an approximation of the performance degradation due to:

1. DRAM controller contention;
2. FSB contention;
3. Shared cache contention;
4. Contention in resources involved in prefetching.

From the classification schemes study, they concluded that the Solo Miss Rate [KBH+08] gives good results while keeping the profiling overhead low. Indeed, the miss-rate of an application is easy to obtain online via hardware performance counters present in commodity hardware. They presented an algorithm exploiting the miss-rate called *Centralised Sort*. The algorithm examines the list of applications, sorted by their miss rates, and distributes them across cores, such that the total miss rate of all threads sharing a cache is balanced across all caches.

The next section discusses cache partitioning as a technique to reduce the inter-thread cache sharing contention.

3.3.2 Cache partitioning

As an alternative to process scheduling, several research studies address cache contention via software-based or hardware-based cache partitioning. Indeed, cache partitioning is a technique that refers to the partitioning of the shared L2 or the L3 caches among multiple computing processes running concurrently on distinct cores. In order to reduce capacity misses experienced on shared caches, the partitioning aims to confine the working set of an application in an portion of the shared cache. Usually, cache partitioning may follow multiple optimisation objectives:

1. Improving overall program performance: improve the IPC of each intervening co-scheduled application by minimising the overall cache miss rate.

2. Providing quality of service (QoS): if we consider that each program has its own performance requirement, an example of QoS can be defined as follows: the performance of a program A when co-scheduled with a program B should never be less than $X\%$ compared to the case when running in solo.

3. Ensuring fairness: an example of a fairness metric is to ensure that the slowdown (speedup) of each co-scheduled program should be identical after cache partitioning. Another fairness metric may follow the optimisation objective of balancing the number of cache misses experienced by each co-runner when sharing a cache.

In the following sections, we discuss the principle of software and hardware cache partitioning techniques. Besides, in order to reduce the inter-thread cache contention, we also present an approach that combines both software and hardware techniques.

3.3.2.1 Software cache partitioning

Most often, software cache partitioning techniques rely on the classical OS-page coloring. In this case, page coloring is implemented inside th OS virtual memory manager or inside virtual

machines. In physically indexed caches, page colouring aims to control the mapping of physical memory pages to a processor's cache blocks. Figure 3.7 illustrates the OS-page colouring technique. Memory pages in the physical address space are assigned three different colors, pages with the same color are mapped to the same cache blocks (a group of cache sets). Page colouring aims to: 1) maximise the total number of physical pages cached by the processor and 2) reduce conflicts by ensuring that contiguous pages in virtual space do not map to conflicting physical pages [TDF90, CJ06, ATSS09, LLD+08, ZDS09]. Therefore, when physical pages are required by an application, the OS will attempt to allocate free pages that are contiguous from the CPU cache's view.

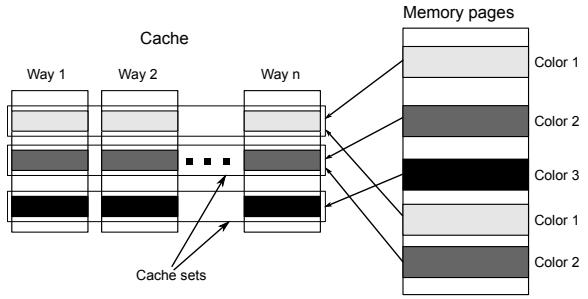


Figure 3.7: An illustration of the page colouring technique [ZDS09]

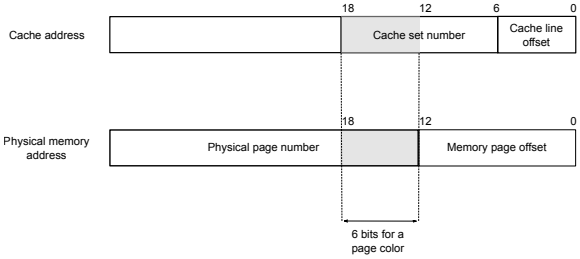


Figure 3.8: Page colors codification on an X86 architecture

As stated above, memory pages that map onto the same cache portion (blocks) are assigned the same colour. The amount of cache size to allocate to each co-running application is controlled by the colour of pages assigned to each application. By doing so, the operating system can isolate the shared cache usage of co-running applications. Assigning a page colour to a physical page works as follows: a physical address contains several common bits between the cache index and the physical page number (See Figure 3.8). These bits are referred to as page colour. The maximum number of colours that a platform can support can be computed by the following formulas:

$$Colors = \frac{CacheLineSize \times NumberofSets}{PageSize}$$

Given this total number of colours, an operating system (OS) can partition the shared cache between the co-running processes by assigning a given number of colours for each of them. The number of colours to provide for each application is dependent on the optimisation objective: performance, QoS or fairness. It can be computed statically (once for each application) after an application profiling phase or dynamically to adapt to program's time-varying phase behaviour. The next section discusses the principle of hardware cache partitioning.

3.3.2.2 Hardware cache partitioning

Hardware cache partitioning techniques mainly focus on efficient (cache sharing aware) cache replacement policies. The goal is to minimise cache misses or maximising fairness. Contrary to OS or programmer approaches, hardware techniques assume that a co-schedule is already determined by the OS or by the programmer, and the hardware's task is to optimise the performance for the given co-schedule by dynamically allocate/partition the shared cache between the running jobs.

As an example of such hardware technique, we present the work of Qureshi *et al.* [QP06]. They proposed a hardware mechanism (UCP for Utility-based Cache Partitioning) to partition

a shared cache between multiple applications. The partitioning depends on the reduction of cache misses that each application is likely to obtain for a given amount of cache resources. The proposed mechanism monitors each application at runtime, it uses a hardware circuit (UMON: utility monitoring) to obtain information about benefit (utility) of cache resource. Mainly, for each application, the mechanism estimates the number of hits and misses for all possible number of ways allocated to the applications. The approach is based on reuse distance profiles. The idea behind the approach is as follows. The utility of a cache resource can be directly correlated to the change in the number of cache misses or improvement in performance of the application when the cache size is varied. Using the utility information, the cache is then partitioned (decide the number of cache ways to allocate for each application) in order to minimise the number of cache misses of the co-running applications.

We presented in the previous sections the principle of software and hardware cache partitioning. The next section discusses an approach that combines hardware and software techniques for shared caches partitioning to reduce the impact of inter-thread cache contention.

3.3.2.3 Combined hardware and OS approach for shared caches management

To illustrate the principle of a combined approach, we present the work of Rafique *et al.* [RLT06]. They proposed an architectural support for an OS to manage shared caches with multiple policies. The idea behind the scheme follows these observations:

1. Shared cache resources are purely managed in hardware with simple replacement policies such as LRU.
2. Managing shared caches in hardware does not offer flexibility to handle all the sharing scenarios.
3. The OS can offer the required flexibility to adapt to different cache sharing scenarios.
4. Managing shared caches purely in software (OS) is impractical due to a high overhead.
5. A combined software/hardware approach may meet the double objectives: performance and flexibility

From the observations presented above, the authors propose a scheme which consists of a hardware cache quota management mechanism, an OS interface and a set of OS level quota orchestration policies. When the OS assigns some portion of the cache (*i.e.* quota) to a given application, the hardware mechanism guarantees that these OS-specified quotas are enforced in shared caches. The quotas are defined by the OS for each co-running application and multiple applications can be assigned to the same portion of the cache. All the assigned quotas are not fixed for the whole application execution life. Indeed, the OS can adapt the quotas to the demand of applications during regularly scheduled OS interventions.

The quota assigned to an application by the OS is specified in terms of number of ways or a set granularity in the cache. The hardware checks that the quota assigned to a process is not violated at the time of cache block replacement. To do so, the hardware needs to know the identity of the running process. The identity of the process is stored in a special register called `SID` (*i.e.* sharer identifier) to identify the identity of the process running on the current processor. Whenever a processor makes a memory request, its `SID` register is used to access a special hardware table `SQT` (*i.e.*, sharer quota table) to relieve the quota value of the process currently running on that processor.

The idea of dynamic partitioning of shared caches was first investigated by Suh *et al.* [SDR02, SRD04]. They proposed an on-line memory monitoring scheme utilising a set of hard-

ware counters. The counters indicate the marginal gain in cache hits as the cache size is increased giving the cache miss-rates for each process as a function of cache size under the standard LRU replacement policy. Using these monitoring information, a partitioning module implemented in software partitions the cache among the active processes so as to minimise the overall cache miss-rate.

3.3.3 Discussion on inter-thread shared cache contention

Cache contention has motivated several studies, going from performance prediction models and process scheduling policies in one hand, and software and hardware cache partitioning techniques on the other hand. Prediction models intend to qualify and quantify the impact of cache sharing on the performance of multi-programmed workloads. In other words, these models study the extent to which cache sharing hurts the performance of co-scheduled applications. In general, the output of such models is the number of extra cache misses when the cache is shared compared to a solo run. In reality, the prediction accuracy is dependent on the complexity or simplicity of the model. If the model is too simple, the accuracy goes down. A complex model will be impractical to implement in real systems. Almost all the studies related to inter-thread cache contention focus on the optimisation of some hand built workloads. Once the prediction models are applied to a workload, the output of such models can be used to propose scheduling policies.

Cache partitioning techniques delivers promising results by confining the working set of each application in some portion of the cache. However, they have some limitations. First, software techniques require non trivial change to virtual memory manager in the OS. Besides, the size of the portion of cache to assign is manipulated in the memory page granularity; the question which may rise: how many page colors to assign for each process? While simple metrics could lead to unpredictable performance behaviour, more sophisticated metrics will be hard to compute on-line. Second, hardware techniques offer better flexibility in terms of the size of cache to assign for each process. This is true because a lot of partitioning policies can be implemented at the cache block replacement. However, these techniques are based on new micro-architectures with no guarantee to be implemented in future processors. Moreover, the impact of such designs is unclear from the needed-hardware to build such machines.

The next section discusses techniques to characterise the amount of inter-thread data sharing for multi-threaded applications and techniques to exploit that sharing by means of thread placement. In the context of multi-threaded applications, adequate thread placement may lead to better performance statbility and to siginificant performance improvement.

3.4 Exploiting data sharing with thread affinity on multicore architectures

Modern shared chip multiprocessors (CMPs) consist of several multicore processors, where each processor has a hierarchy of memory caches. This implementation design allows to exploit data sharing between threads running on such platforms. Of course, to exploit that, a multi-threaded application has to meet two conditions. First, the application's threads have actually to access or to share common data. Second, the reuse distance has to be sufficiently short to effectively exploit these shared data across multiple threads (see Section 3.2 for more information about data locality measurement and prediction). In this context, *thread affinity* in multicore proces-

sors (see Figure 3.9) can be defined as the process of assigning each thread (process) to one or a subset of cores. The idea behind this definition is to impose to the OS to run a given thread on core #N or to run that thread on all the cores except cores #0 and #1 for example. The operating system takes into account this notification, and the thread runs only on the allowed cores.

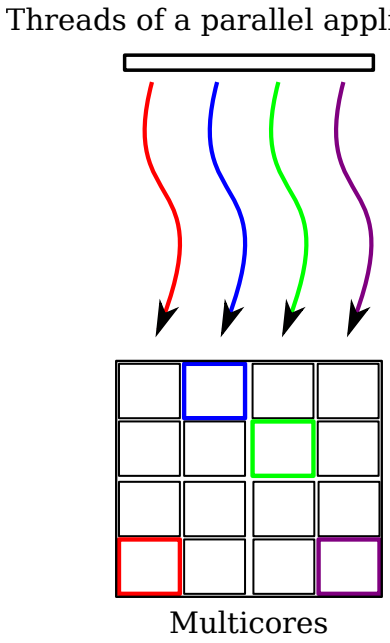


Figure 3.9: Thread affinity in multicore processors

There are many advantages for thread affinity. The first one is enhancing the inter-thread data locality. For instance, if two threads make extensive accesses to common data in memory, it is better to place them on adjacent cores sharing the same L2/L3 cache, or the same NUMA node. Doing so, we would decrease the number of cache misses. Indeed, if one thread brings the required data to some cache level, the second thread accessing the same data element will avert a cache miss. Thus, the latency of memory access is reduced. Furthermore, binding threads to cores by considering the machine architecture may help hardware prefetching of frequently accessed shared regions. Enhancing data locality is another benefit from thread affinity. If we consider a machine with two multicore processors and a memory bound application creating two threads, it would be preferable to bind each thread to a distinct processor socket. First, such thread placement will enhance the single thread data locality of both threads. Second, it reduces the cache access contention. Consequently, it leads to better cache performance. On the other hand, wrong thread placement can lead to a severe performance degradation. Indeed, the overhead of accessing common data between two threads running on distinct cores depends on their physical location. Therefore, thread affinity to cores called also *thread pinning* is of high importance.

In the absence of an explicit management of thread placement by the application programmer, the decision about thread placement is achieved by the operating system (OS) scheduler or by the runtime library. Unfortunately, current OS consider every core as a distinct processor. If we have a processor with, say 8 cores, the OS sees 8 homogeneous processors that are capable of executing concurrent threads, processes or jobs. However, in terms of performance tuning, we cannot consider the cores as homogeneous because they share common micro-architectural

resources: L2 or L3 shared caches, shared memory buses, etc.

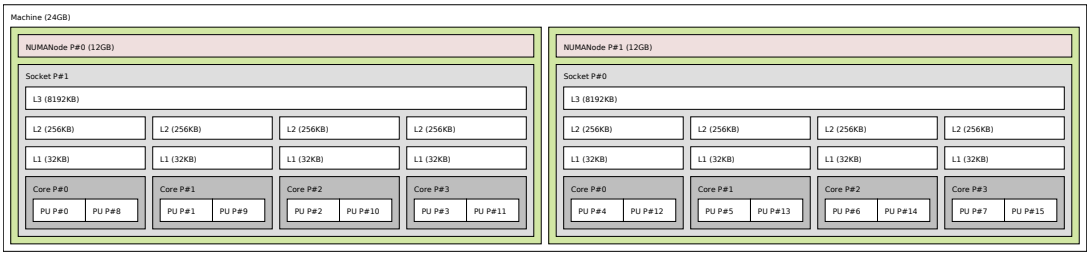


Figure 3.10: `Nehalem` NUMA machine architecture

In order to illustrate the program performance of a parallel application running with multiple threads that access common data in function of different thread placements (different cache sharing situations), we use a synthetic benchmark. Figure 3.11 reports the performance of a synthetic micro-benchmark (Listing 3.1) depending on multiple cache sharing situations. The micro-benchmark creates two OpenMP threads, each increments a shared global counter concurrently. The test machine is a NUMA machine with two `Bloomfield` sockets (`Nehalem` micro-architecture). Each processor has 4 cores with a shared L3 cache. The platform has two L3 caches of 8 MB, one on each chip, for both instructions and data. The main memory size is 12GB. Each chip in the platform features an integrated memory controller. The `hyper-threading` and the `Turbo Boost` technologies were enabled (the diagram of the machine is given in Figure 3.10). We have to notice the following: first, since Hyper-Threading is enabled, hardware threads (HWTs) 4 and 12 share an L2 cache. Second, HWTs 4, 5, 6 and 7 share an L3 cache. Finally, The HWT 4 in one side and HWTs 0, 1, 2 and 3 on the other side are on distinct NUMA nodes. Figure 3.11 reports the speedup of the median execution time (35 runs for each software execution) for various cache sharing configuration relative to the C1 configuration. It is clear from the figure that, more the distance (in termes of thread placement) between threads is important, more the latency of access to shared data is important. Consequently, program performance is highly sensitive to thread placement.

Listing 3.1: OpenMP micro-benchmark code

```
#pragma omp parallel default(none) private(i) shared(N,counter)
{
    #pragma omp for
        for(i=0; i<N; i++) {
            #pragma omp critical
                sum += i;
        }
}
return counter;
```

The next section discusses some system calls and libraries to manage thread affinity explicitly in applications code.

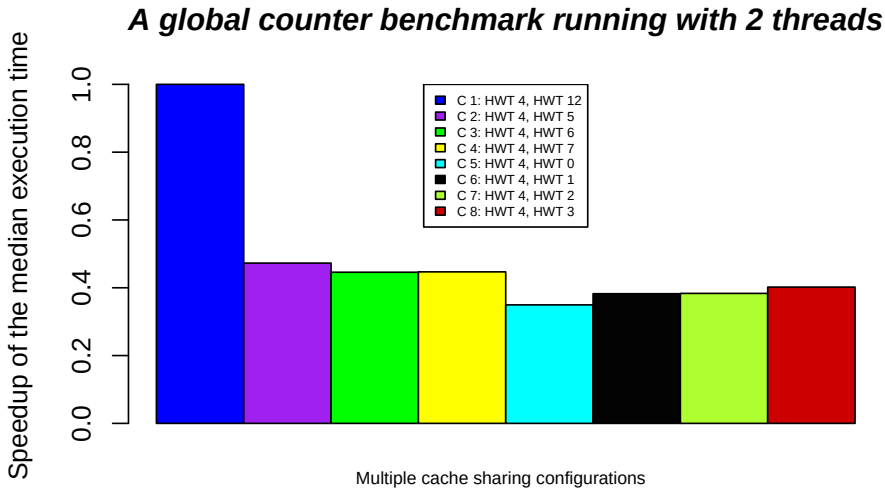


Figure 3.11: Micro-benchmark program performance depending on thread thread

3.4.1 Explicit software support for thread affinity

Almost all major operating systems (except MAC OSX) propose programming interfaces to set thread affinity. The programmer can decide on which core a given thread should run. Unfortunately, it is challenging for a programmer to determine which are the shared data regions and the intensity of sharing between threads at development time. There is an another problem with this approach, the required effort to rewrite each application to benefit from this service can be important. Other problems can rise as well: multiple source code files, shared libraries, etc. Regarding the Linux kernel, one can use the `cpuset` interface. The `cpuset`[7] interface provides a mechanism for assigning a set of CPUs and memory nodes to a set of processes (threads). `Cpusets` constrain the CPU and memory placement of processes/threads to only the resources within a task's current `cpuset`. The OS scheduler will not schedule a task on a CPU that is not allowed in its `cpus_allowed` vector, and the kernel page allocator will not allocate a page on a node that is not allowed in the requesting tasks `mems_allowed` vector.

As explained earlier, by using `cpusets`, the programmer can mainly control the list of CPUs and the memory allocation policy (will be detailed later) of a particular task. The different system calls provided by the Linux kernel to manipulate this interface are as the following:

1. **CPU affinity**. The CPU affinity interface allows the programmer to manage the process's (thread) CPU affinity mask. The mask determines the set of CPUs on which it is allowed to run. Setting and getting the process's CPU affinity mask can be achieved by the `sched_setaffinity` and `sched_getaffinity` system calls. The `Native POSIX Thread Library` (NPTL) implements also a similar interface. It allows to confine a given thread to a set of CPUs. Similarly, this can be achieved by the functions `pthread_setaffinity_np` and `pthread_getaffinity_np`.

2. **Memory affinity**. The memory affinity interface allows the programmer to set the memory policy of its application. The programmer can use the `mbind`, `set_mempolicy` and

[7] A `cpuset` interface is exposed by the OS as a file system which can be accessed by special system calls

`get_mempolicy` system calls. The `libNUMA` library [Kle05] offers a simple programming interface to the NUMA (Non Uniform Memory Access) policy supported by the Linux kernel. Basically, this library is a wrapper layer over the system calls. Available policies are page interleaving (*i.e.* allocate in a round-robin fashion from all or a subset of the nodes on the system), preferred node allocation (*i.e.* preferably allocate on a particular node), local allocation (*i.e.* allocate on the node on which the thread is currently executing), or allocation only on specific nodes (*i.e.* allocate on some subset of the available nodes).

The next section presents some related work on inter-thread data sharing detection and its exploitation by thread placement at the application level.

3.4.2 Application level data sharing detection and thread mapping

In the field of quantifying the importance of exploiting data locality between threads, Bellosa *et al.* [BS96] examined the performance implications of locality information usage in thread scheduling algorithms for shared-memory multiprocessors. They proposed a non preemptive user-level thread package with an application interface to inform the runtime system about memory regions repeatedly used. These hints are used to trigger prefetch operations at each process scheduling decision to hide memory latency. In addition, they proposed scheduling policies based on locality information of individual threads derived from hardware performance counters. The data locality information needed by the proposed algorithms consist of cache miss rate, the processor stall time, and the processor that was assigned to the process during its last execution. They focused on enhancing cache reuse of individual threads not multi-threaded applications. In order to enhance the chance of cache reuse, Bellosa [Bel97] proposed to schedule sequentially (*i.e.* one after each other) kernel threads that share large parts of memory. He proposed the use of TLB information to detect memory pages sharing between threads.

Similarly, Weissman [Wei98] proposed an approach for improving data locality. It uses hardware performance counters and program-centric code annotations to guide thread scheduling on symmetrical multiprocessors (SMPs). The idea behind this approach is the use of an analytical model which takes the number of cache misses and source code annotations as input; the later are used to express the sharing patterns inherent in the applications. The output of the model is a prediction of threads footprints. Using this model, he proposed some practical scheduling policies to enhance the locality of applications.

Zhang *et al.* [ZJS10] conducted a measurement analysis to study the influence of chip multiprocessors (CMP) cache sharing on multi-threaded performance applications using the PARSEC [BKSL08] benchmark suite. Through measurement, they tested various factors of interactions between cache sharing and the performance of multi-threaded applications such as types of parallelism (data-level or pipelined), input datasets, numbers of threads and the assignment of threads to cores. They suggested that cache sharing has very limited influence on the performance of the PARSEC applications due to the large working sets and to the limited inter-thread data sharing of the tested multi-threaded programs. However, they do not conclude that cache sharing has no potential to be explored for multi-threaded programs. Regarding the PARSEC benchmark suite, the authors concluded that current multi-threaded applications are not well optimised to leverage the power of existing chip multiprocessor (CMP) architectures.

Tang *et al.* [TMV+11] studied the impact of sharing memory resources on data-centre applications. Across these applications, they investigated the importance of thread to core pinnings to share or to not share caches and bus bandwidth. Through measurements, they also in-

vestigated the impact of co-running threads from multiple applications with diverse memory behaviours to discover the best pinning suitable to the co-running applications. By studying some key performance characteristics of the applications when running alone and by using hardware performance counters, they proposed a heuristic approach to compute the best pinning of threads when co-running multiple independent applications.

Kazempour *et al.* [KFA08] examined the performance effect of exploiting cache affinity on multicore multiprocessors[8] and uniprocessors[9]. They demonstrated that, while exploiting cache affinity on multicore uniprocessors has no measurable impact on performance, performance improvement from exploiting cache affinity on multicore multiprocessors is significant.

Terboven *et al.* [TaMS+08] examined the programming possibilities to improve memory pages and thread affinity in OpenMP applications running on ccNUMA architectures. They provided a performance analysis of some HPC codes which may suffer from ccNUMA architectures effects.

Binding threads in nested OpenMP parallel regions is a challenge. Indeed, there is no guarantee that two active parallel regions with identical ancestry will be executed by the same set of system threads. Schmidl *et al.* [STaMB10] discussed the performance problems of nested OpenMP programs on ccNUMA machines. They provided a library to retrieve the hardware information (cache topology) of the target machine and to set a static thread binding strategy for each parallelisation level in the OpenMP program. They used the `OPARI` OpenMP instrumentation tool to add a function call to their library at the beginning of every parallel region to find out which threads are used. The library does not detect any sharing behaviour between threads, it just applies some predefined thread binding strategies to the encountered nested OpenMP parallel regions.

Klug *et al.* [KOWT11] proposed a framework to automatically determine the thread pinning best suited for a multi-threaded application based on hardware performance counters information. The idea behind the framework is to evaluate the performance (measured by the CPI) of a set of different thread pinning strategies for a fixed quantum of time and select the strategy with the best CPI. The framework requires that the time measurement interval and a set of multiple pinning be provided as input.

Marathe *et al.* [MTM10] proposed a hardware-assisted page placement approach based on automated tracing of the memory references made by application threads for ccNUMA machines. The objective is to allocate pages near processors that most frequently access that memory pages. During trace generation, hardware performance counters are used to extract an approximate trace of memory accesses. The target program is run for one stable execution phase of the program and data trace is collected (cache misses and TLB information). The stable execution phase must be manually identified by the user. The idea is to collect a snapshot of the program's memory access patterns during a snippet of its stable execution phase, which becomes the basis for guiding page placement decisions. The collected trace data is used to compute the page affinity, *i.e.* the node to which the page is bound and the entire program is re-run using this data trace. The approach is based on the first-touch page placement policy.

Song *et al.* [SMD07,SMD09] uses a feedback guided method to compute thread affinity. The

[8]Multiple processors, each with multiple cores.
[9]Single processor with multiple cores

method relies upon binary instrumentation to acquire the memory sharing relationship between user-level threads by analysing the memory trace. After, they build an affinity graph to model the relationship. Then, they used hierarchical graph partitioning to compute optimised thread schedules. They also introduce an analytical model to estimate the cost of running an affinity-based thread schedule. Their model considers the number of addresses accessed in common, not the number of access to common memory line addresses. In fact, considering only number of addresses accessed in common does not reflect the real cache access intensity. Actually, the approach that we follow in Chapter 5 to compute thread pinnings is quite close to that of [SMD07]. However, our work differs from theirs in four main points: 1) we take into account the performance variability (running each application 35 times) when we run a given benchmark with different thread affinity schedules, 2) we focus on real complex applications and not on synthetic benchmarks or small kernels, 3) we use linear programming and graph partitioning to compute optimised thread pinnings (produce optimal results as far as cache performance is concerned) and finally 4) our work is not limited to find the best schedule against the default one of the application; instead, we investigate how the overall performance of a given multi-threaded applications behave under a set of predominant thread affinity schedules. The last point is important because we show that simple strategies (do not need any memory tracing phase) perform very well in many OpenMP applications .

Jeannot *et al.* [JM10] proposed an algorithm that maps MPI processes to cores in order to reduce communication cost of the whole application. The described algorithm requires the target's application communication pattern. This pattern consists of the global amount of data exchanged between each pair of precesses in the MPI application. The later is stored in a communication matrix. To retrieve the communication pattern, they instrumented low-level communication channels in the MPICH2[10] MPI implementation to track point-to-point and collective communications. The approach needs two runs, the first run for data collection and computing the processes mapping and re-run of the target application with the computed processes mapping. They concluded that although the proposed algorithm outperforms some placement strategies that do not require profiling (heuristics), there is a slight performance difference between them. As an explanation for this performance behaviour, the authors suggested that may be it is due to modelling issues; as the communication matrix is an aggregated view of the whole execution and does not account for different phases of the application with different communication patterns.

The next section presents some related work on compiler and runtime level management of inter-thread data sharing and thread placement.

3.4.3 Compiler and runtime data sharing detection and thread mapping

Some studies have addressed the data cache sharing at the compiler or runtime/OS level. These studies have focused on improving data locality in multicores by being aware of the architecture topology.

Sridharan *et al.* [SKM+06] proposed to exploit the locality of the critical section data. The principle of this approach is simply to enforce an affinity between locks and the processor that has cached the execution state of the critical section protected by that lock. They also investigated the idea of migrating threads to the processor that has cached the highly-contended lock. The proposed technique heavily relies on the kernel thread scheduler. Furthermore, it requires

[10]MPICH2 is an open source implementation of MPI.

that the scheduler must be able to identify the thread that currently holds a contended lock. The identification is achieved by annotating user-level synchronisation libraries and a kernel scheduler modification.

Tam *et al.* [TAS07] proposed thread clustering to schedule threads based on data sharing patterns detected online using hardware performance monitoring counters. The mechanism was implemented inside a Linux operating system running on IBM `Power5` multiprocessor. For testing purpose, they concentrated on commercial multi-threaded server programs. Iteratively, they attempted to group threads exhibiting high degree of data sharing in the same processor. Using hardware performance counters, CPU cycles are broken down and assigned to different microprocessor components to determine the performance impact of cross-chip communication. They monitored the addresses of cache lines that are invalidated due to remote cache-coherence activities and build a data structure for each thread. Each data structure shows which data items each thread is fetching from caches on remote chips. After that, they compared these data structures to detect the sharing behaviour between threads and cluster them accordingly.

Broquedis *et al.* [oBFG+10] proposed an OpenMP runtime for NUMA architectures. The runtime is based on a multi-level thread scheduler and on a NUMA-aware memory manager. The user can specify to the runtime information about thread and memory pages placements. These information are converted by the runtime into scheduling hints related to thread-memory affinity issues. These hints enable dynamic load distribution of threads and data over NUMA architectures.

Lee *et al.* [LWRC10] proposed a framework to automatically adjusts the number of threads in an application to optimise system efficiency. The framework uses an off-line analysis to estimate what type of threads will exist at runtime and the communication patterns between them. Using this information and using graph partitioning algorithms, the framework dynamically combines threads. The tested applications were compiled statically to spawn 128 threads at runtime. The framework was prototyped using the Low-Level Virtual machine (LLVM) toolsets for compiling and running the applications. The work assumes a uniform distribution of the data between threads.

Kandamir *et al.* [KMN+09, KYM+10, ZKY11] discussed a compiler directed code restructuring scheme for enhancing locality of shared data in multicores. Using a source-to-source compiler, the scheme operates as follows. First, the arrays accessed by the application are divided into equal size data blocks. Second, for each core, the set of loop iterations assigned to it are divided into equal size computation blocks. Finally, the compiler captures data dependences and data sharing between computation blocks which are assigned to cores. The goal is to increase data reuse regarding the access to data blocks by the computations blocks.

Our work differs from the last efforts in two main points. First, we dot not focus on providing a new thread scheduling strategy. Unlike other studies, we perform statistical performance evaluation (running multiple times, we fix the experimental setup, etc.). We aim to study the impact of different thread affinity strategies on performance stability as long as as data sharing is concerned. Moreover, we consider also NUMA effects, this is not actually the case for most of the previous studies. Second, when it comes to compute a scheduling affinity, we rely on a profile-guided method. Using dynamic binary instrumentation, we fully analyse optimised binaries regardless of the compiler. Furthermore, we believe that extracting all data dependencies and data sharing at compile time may not be sufficient, because these information

depend on the working set which is known only at runtime.

3.4.4 Discussion about inter-thread data sharing and thread placement

Data sharing characterisation is inherently dependent on an accurate inter-thread data locality or data reuse study. Indeed, in terms of performance improvement, converting memory accesses to shared memory lines in shared caches requires to qualify and quantify the amount of sharing implemented in the multi-threaded application, and implement a thread placement strategy to exploit that propriety. Knowing that, an application that has short reuse distances means that it worth the effort to implement a data sharing strategy exploitation. On the other hand, long reuse distances may necessitate to rewrite the application in order to shorten these reuse distances, thus, effectively exploit that sharing behaviour. There is another category, it consists of multi-threaded applications in which the access to shared data between threads is rare. Therefore restructuring the application may be useless.

In the case of existing shared data, thread affinity offers a simple approach to transform data sharing into performance improvement. However, from the performance perspective, cache sharing is tricky. With a wrong pinning, instead to be beneficial, it can degrade performance (see Section 3.3), or it can lead to performance instability (see Section 3.1). In addition, an affinity strategy that focuses on data reuse only in order to improve cache performance may not be sufficient. Indeed, an efficient affinity strategy has to consider other factors (*i.e.* data prefetching, memory pages placement, the workload of the machine, etc.).

The quantification of data sharing and its corresponding affinity strategy can be achieved mainly in three ways. First, data sharing is quantified by the programmer (application level) and an adequate affinity is applied to the application for its lifetime. This approach has a main drawback, it is not clear or easy for the application programmer to quantify the amount of sharing implemented in the application. Second, a compiler may perform some static analysis to deduce the amount of sharing implemented in the application. A compile time approach has the advantage to be automatic and it does not require the programmer intervention. However, data sharing quantification at compile time may not be accurate. Many reasons contribute to this inaccuracy. Most importantly, it is not obvious to capture all memory references at compile time for programs with irregular access patterns or for programs that make heavy use for pointers. It is also possible to consider the number of threads and the data input which are usually known at runtime. However, the impact of the two later factors can be limited by a parametric analysis for instance. Finally, we can consider a runtime approach. The advantage of this approach is its ability to adapt to the runtime environment (*i.e.* the workload of the machine, number of available cores). Despite this, the profiling/characterisation overhead of the application may not be negligible.

We discussed in this chapter multiple aspects related to enhancing program performance in multicore architectures. The next chapter presents an experimental study to quantify and qualify the variability of program execution times in multi-threaded programs.

Chapter 4

Measuring and Analysing the Variations of Program Execution Times on Multicore Platforms

In this chapter, we present a study of variations of program execution times. We show that while these variations are statistically insignificant for large sequential applications, we observe that parallel native OpenMP programs have less performance stability. We investigate multiple factors such as thread affinity and memory pages size in the goal to quantify the influence of that factors on the variability of execution times.

4.1 Introduction

Multicore architectures are nowadays the state of the art in the industry of processor design for desktop and high performance computing. With this architectural design, multiple threads can run simultaneously exploiting a thread level parallelism. Unfortunately, achieving better program performance is a little bit hard work. Indeed, programmers have to deal with some issues in both software and hardware levels (thread and process scheduling, memory management, shared resources managements, energy consumption and heat dissipation of cores, etc.). Furthermore, the lack in understanding the interactions between the operating system layers, applications and the underlying hardware makes this task even more difficult. A good understanding of these interactions may be exploited in performance evaluation, compiler optimisations and in process/thread scheduling to achieve a better performance stability, reproducibility and predictability.

In this context, applications designers and performance analysts have to iteratively investigate how to achieve the best performance and checking the behaviour of their applications on that architectures. Most often, program execution time is considered as the first metric to investigate in the process of performance evaluation. The execution time is usually observed by measurements, or can be simulated or predicted with a performance model. In our thesis we consider direct measurements (either by hardware performance counters, or by OS timing functions calls). Contrary to emulated or virtualised programs (such as Javabyte-codes), native program binaries are executed directly on the hardware with possibly some basic OS requests (OS function calls). Our current study focuses on this family of programs: we consider the sample of SPEC 2006 and SPEC OMP2001 [Sta06] benchmark applications. We do not consider binary virtualisation or byte-code emulation because they add software layers influencing the

program performance in a more complex way: garbage collector strategies, threads organisation, caching and dynamic compilation techniques all may dramatically influence the measurements of program execution times. Direct measurements of native applications have one software layer (namely the OS) between the user code and the hardware. Unfortunately, the measurement process may also introduce errors or noise (the act of measuring perturbs the program being measured) that can affect our experimental results. For example, there is a time required to read a timer before the code to measure and store the timer after this code. Experimental setups may also introduce other factors (see Section 3.1.1) lead to variations of program execution times. Thus, if we execute a program N times, we may obtain N distinct execution times.

For our study, we introduce some experiments aimed to measure, quantify and analyse the variations of program execution times on an Intel multicore machine. We report measurement results for single-threaded applications (SPEC CPU2006), as well for parallel multi-threaded applications (SPEC OMP2001) with a fixed data input. The parallel applications use the OpenMP paradigm, one of the most used in parallel programming model on shared memory computers. We show that large SPEC CPU2006 applications have minor variations with the train data input. This of course does not guarantee that the variations of sequential applications would always be negligible especially for small codes (kernels). Unlike single-threaded applications, we show that the variations of execution times of OpenMP applications are really sensitive from a human user point of view.

This chapter is organised as follows. Section 4.2 introduces the experimental setup and methodology that we follow. Section 4.3 studies the performance variability of sequential applications (SPEC CPU2006), and parallel OpenMP applications (SPEC OMP2001). Section 4.4 studies the impact of thread placement SPEC OMP01 program execution times. The influence of background co-running processes on the performance of SPEC OMP2001 is studied in Section 4.5. Finally, the influence of co-running processes on the performance of OpenMP micro-benchmarks is studied in Section 4.6 before concluding.

4.2 Experimental setup and methodology

4.2.1 Hardware setup

As an example of hardware machine, we use an Intel (Dell) server with two `Clovertown` processors. Each processor has 4 cores, while each couple of cores have a shared level 2 cache. Our system has two L2 caches on each chip with 4 MB, for both instructions and data. The core frequency is 2.33 GHz. The maim memory size is 4 GB RAM. The frontside bus has a clock rate of 1.33 GHz. The main features of the test machine are summarised in Figure 4.1.

4.2.2 Software environment

The version of the Linux kernel is X86_64 2.6.26, patched with `perfmon` kernel 2.81. We used multiple compilers: `gcc` 4.1.3, `gcc` 4-3.2, `icc` 11.0 and `ifort` 11.1, all applied with optimisation level `-O3 -fopenmp`. The experimented benchmarks are SPEC CPU2006 and OMP2001 applications run with the train input and various configurations of thread numbers. We also designed our own micro-benchmarks to analyse the interaction between the software, the micro-architecture and the OS layer.

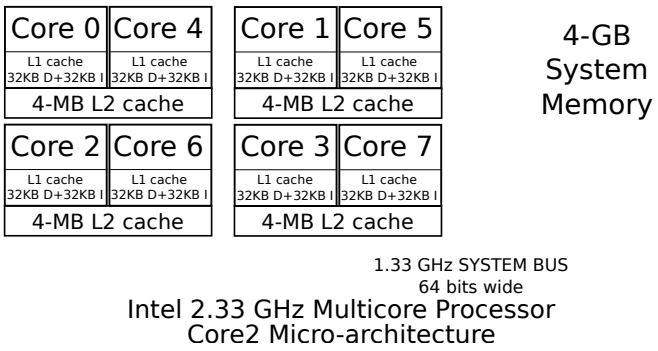


Figure 4.1: Dual processor architecture

4.2.3 Experimental methodology

In order to improve the reproducibility of the results, the experiments were done following some practices:

- The data input is fixed.
- The test machine was entirely dedicated during the experiments to a single user.
- Running each benchmark 31 times [Raj91, TWB10] for each software configuration. This high number of runs allows us to report statistics with a high confidence level;
- Unset all the shell environment variables that were inessential;
- The experiments were done on a minimally-loaded machine (disable all inessential OS services except `sshd`);
- Starting address of the stack randomisation deactivated (this is an option in the Linux kernel versions since 2.6.12);
- Dynamic voltage scaling (DVS) disabled;
- Using the build system and scripts of SPEC CPU2006 and OMP2001 to compile and optimise applications, launch them, measure execution times, check validity of the results and report the performance numbers;
- The SPEC system measurement of execution times relies on the `gettimeofday` function;
- The successive executions are performed sequentially in back-to-back way;
- No more than one application was executed at a time, except when we study co-running effects.
- We use violin plots to report the program execution times of the 31 execution of each software configuration (see Section 4.2.3.1 for more details).
- All observed execution times are reported, we do not remove any outliers (except if the run crashes or produces a wrong output result)

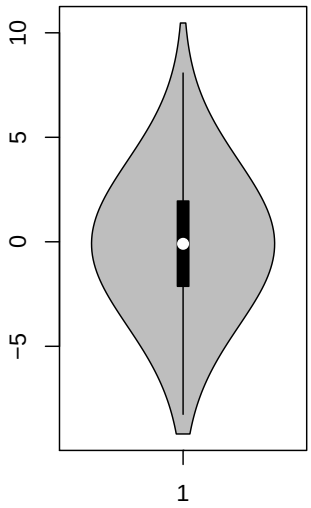


Figure 4.2: A violin plot example

4.2.3.1 Reporting performance data with violin plots

Violin plot gives a good and simple graphical way to check whether an application exhibits or not variability of program execution times. The Violin plot is similar to box plots, except that they also show the probability density of the data at different values. Figure 4.2 shows an example of violin plot. The white dot in each violin gives the `median` and the thick line through the white dot gives the inter-quartile range.

4.2.4 Definition of program performance variability

When we observe a sample of execution times of an application P, say $\{t_1, \cdots, t_n\}$ where t_i is the execution time of the i^{th} run, then we may define the variability according to many metrics. Any used metric must define the *feeling* of the end user about the instability of the execution time of the application. We can use the usual sample variance, or $\frac{|\max_i t_i - \min_i t_i|}{\bar{t}}$ where $\bar{t}$ is the sample mean, or $\frac{|\max_i t_i - \min_i t_i|}{med(t)}$ where $med(t)$ is the sample median. In our study, we use metrics that measure the disparity between extrema observations (outliers):

1. An absolute variability, which is the difference between the maximal and the minimal observed execution times $AV(P) = |\max_i t_i - \min_i t_i|$;

2. A relative variability, which is the absolute variability divided by the maximal observed execution time $RV(P) = \frac{AV(P)}{\max_i t_i} = \frac{|\max_i t_i - \min_i t_i|}{\max_i t_i}$.

Now the question is how to decide about a definition of a program with non negligible performance variability. Since any experimental measure brings a sample variation (it is impossible in practice to observe exactly equal execution times), when can we speak about non negligible variability? In our study, we say that a program P has non negligible performance variability if its relative variability exceeds 1% ($RV(P) > 1\%$). Another definition may exist; In our context

we chose the previous definition in order to be close to the feeling of a user executing a program interactively (*i.e.* when he launches the program and he waits for its termination).

In the remainder of this thesis, we refer to program execution times variability as the relative variability (RV) multiplied by 100. This will present the variability as a percentage. The next section shows that the execution times of long running sequential applications have marginal variability.

4.3 Study of the variability of SPEC benchmarks execution times

This section presents experiments which aim to study the variability of program execution times of SPEC CPU2006 and OMP2001 applications. While the former are sequential applications, the later are parallel programs written with the OpenMP API. For SPEC CPU2006 benchmarks, we test the relation between the UNIX shell environment size and the variation of program execution times. Following the methodology explained in [MDHS09], we varied the size of the UNIX shell environment. We also experimented two code optimisation level `-O2` and `-O3`. For SPEC OMP2001 applications, we fixed the UNIX shell environment, and we varied the number of threads as: sequential (without OpenMP), 1 thread (OMP version with a unique thread), 2, 4, 6 and 8 threads[1]. The idea behind OpenMP experiments is to study the impact of two factors. First, are the parallel execution with different number of threads lead to variability of program execution times? Second, compare the benefit of the parallel execution with different number of threads against the sequential version.

4.3.1 Variability of SPEC CPU2006 execution times

We used the `gcc 4.1.3` compiler with the `-O2` and `-O3` optimisation flags. We had to add the `--fno-strict-aliasing` option for the `perlbench` benchmark because of a technical error in that code[2].

Figure 4.3 reports the execution times of four applications, and for each Unix shell environment size using violin plots (see Section 4.2.3.1). The leftmost point of the X-axis is for a Unix shell environment size of 0 bytes (the null environment); we generated the data using the `bash` shell and for each point, we added 63 bytes to the environment. The width of a violin plot at y-value y is proportional to the number of times we observed y. Figure 4.3 says that for each UNIX shell environment size in the X-axis, the Y-axis reports the 31 execution times.

From all these figures we can deduce that: 1) the size of the Unix shell environment may influence the execution times and 2) the variations of execution times are minor (less than 1%). These observations are valid for all the SPEC CPU2006 benchmarks that we experimented. Figure 4.4 reports the confidence interval of the mean of these benchmarks. We can see that these intervals are sufficiently tights. These figures show that the sample mean at each Unix shell environment size does not vary in a significant way.

[1] We limited the number of threads to 8, because our experimental machine have a maximum number of cores equal to 8.

[2] The benchmark has some known aliasing issues. Hence the compilation with high optimisation level will most likely produce binaries assuming strict aliasing. The problem was reported in the SPEC CPU2006 documentation.

From the experiments presented in this section, we deduce in overall that varying the Unix shell environment size has a negligible impact (less than 1%) on the variability of the execution times of SPEC CPU2006 benchmarks. This observation is true, whatever the optimisation flag we used (`-O2` or `-O3`).

The next section shows the performance variability of the multi-threaded SPEC OMP2001 benchmarks given a fixed experimental setup.

4.3.2 Variability of SPEC OMP2001 execution times

For SPEC OMP2001, we used the `gcc 4.3.2` and `icc 11.0` compilers. For each application, we generated two compiled binary codes. The first one is sequential or single-threaded (using the `-O3` compilation flag). The second one is a multi-threaded version, it is generated by setting `-O3 -fopenmp` and `-O3 -openmp` compilation flags respectively for the `gcc` and `icc` compilers[3].

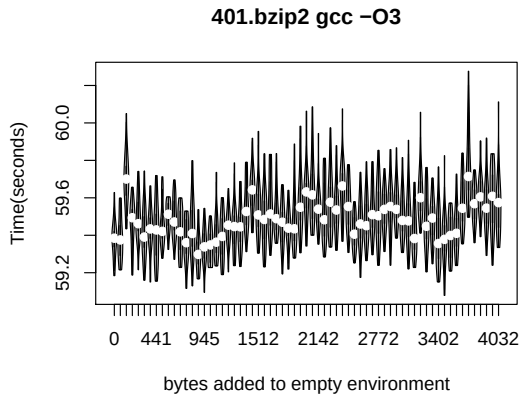
We use violin plots to report in Figure 4.5 the execution times of each application compiled with `gcc`. The UNIX shell environment size was fixed. We choose three applications that highlight significant performance variability. The X-axis represents the different software configurations for the application: sequential version (no threads), OMP version with 1 thread, 2 threads, 4 and 8 threads. The Y-axis represents the 31 observed execution times for each software configuration. We conclude the following observations:

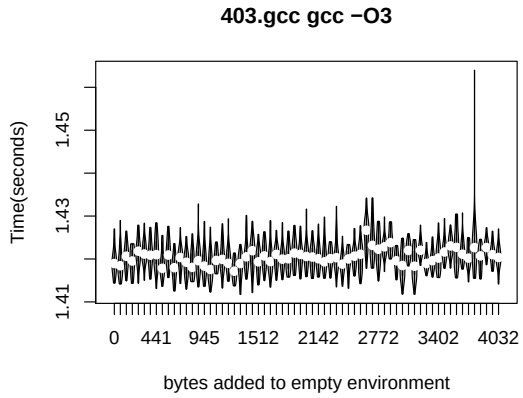
1. The sequential and the single threaded versions do not exhibit significant variability.

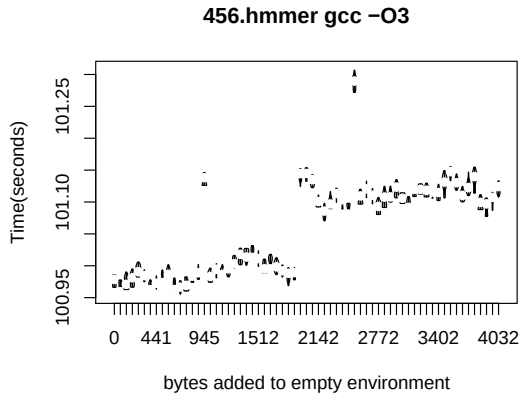
2. When we use thread level parallelism (2 or more threads), the execution times decreases in overall but with a significant disparity. Consider for instance the case of `swim` in Figure 4.5. The version with 2 threads runs between 76 and 109 s the version with 4 threads runs between 71 and 90 s. This variability is also present when `swim` is compiled with `icc`, see Figure 4.6. The example of `wupwise` in Fig. 4.5 is also interesting. The version with 2 threads runs between 376 and 408 s, the version with 6 threads runs between 187 and 204 s. This disparity between the distinct execution times of the same program with the same data input cannot be justified by *accidents* or experimental hazards. Applying the Shapiro-Wilk normality check on performance data we concluded that the execution times are not normally distributed, and frequently have a bias.

3. The case of the application `galgel` is also interesting. In addition to the variability of the execution times for each software configuration, we observe that the performance of the program substantially decreases when increasing the number of threads! This example illustrates that, on a multicore architecture, increasing thread parallelism may bring severe performance loss. We checked the situation of `galgel` when we use the Intel `icc 11.0` compiler instead of `gcc`, and the situation was radically different, see Figure 4.6: increasing the number of threads decreases the execution times. We can observe a huge difference between the performance of the program compiled with gcc vs. icc, either in terms of execution times and in terms of variability. We have to notice that using the `gcc-4.4.3` version of the `GNU` compiler has effectively reduced the execution times when we increase the number of threads (see Figure 4.7). This situation illustrates that the quality of the code generated by a compiler has a significant impact on performance stability.

[3] `gcc` was not able to compile the OpenMP version of `mgrid_m` because of a bug (Bugzilla Bug 33904). The parallel execution of `gafort_m` failed because of a segmentation fault (this execution error was also reported if we use the Intel `icc` compiler).







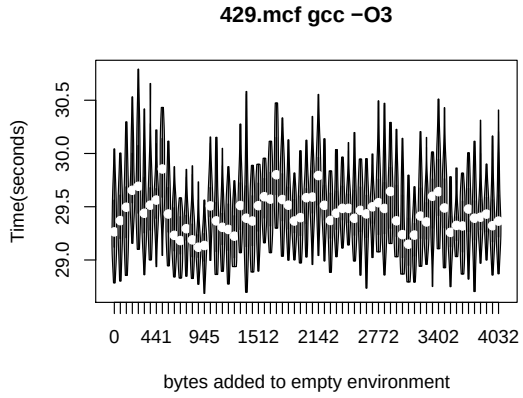
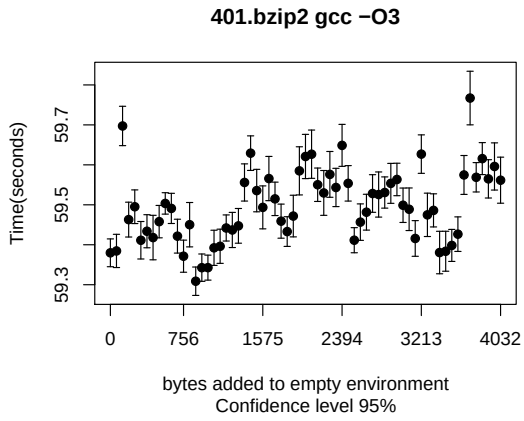
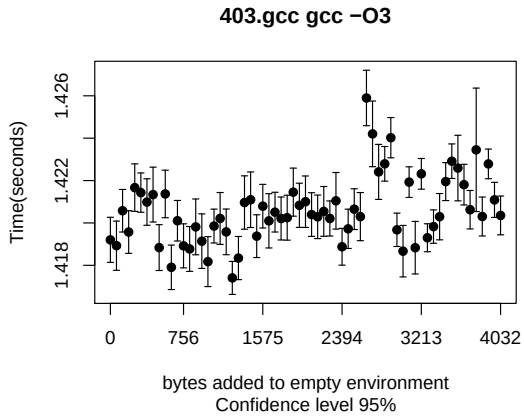
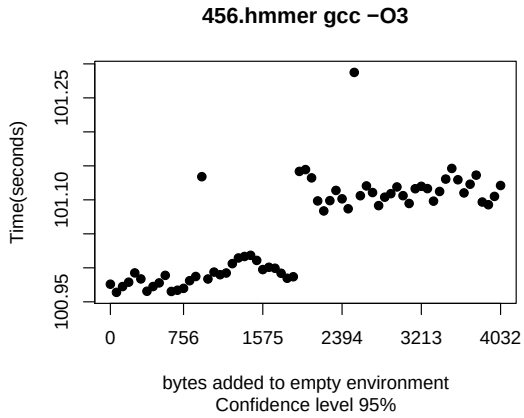


Figure 4.3: Observed Execution Times of some SPEC CPU 2006 Applications (compiled with `gcc`)







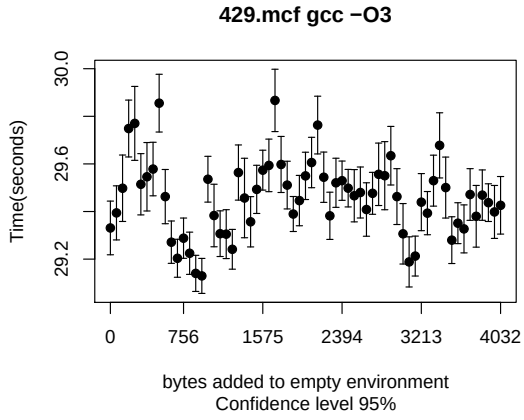


Figure 4.4: Mean 95% Confidence Interval of some SPEC CPU 2006 Applications (compiled with `gcc`)

4. The `galgel` application compiled with the `gcc` compiler, shows that speedup computation is not fair if we consider the minor execution time. We can see from the Figure 4.5 that the violin plots of the second (PAR (1TH)) and third (PAR(2TH)) configurations gives an interesting result on how we have to summarise the performance data of one configuration to single number. If we use the *min* function to summarise data of the two configurations, then, we can say that the third configuration is better than the second one. But if we take the *median* function to summarise these data, we may conclude that the two configurations are similar. We note that the choice of which function to use to define the execution time is crucial and may lead to misleading conclusions about the real behaviour of the system.

5. Figure 4.8 shows that the sequential version of `ammp` is better than its parallel version when parallelisation is achieved with: 1 thread by about 25%, and 2 threads by about 15%. The case of `ammp` shows that the OpenMP API does not necessarily produce faster codes against the sequential version. Although we do not checked the reason for this performance behaviour, we can consider aspects like synchronisation primitives overhead, work imbalance between threads or the compiler makes better optimisations when OpenMP is not enabled (less system calls, less function calls, etc.).

6. When the number of threads is equal to 8, then the variability is significantly reduced on the 8 cores machine.

The next section presents a study of the effect of running SPEC OMP applications with an affinity to the system cores taking into account the impact of sharing the last level cache (L2 cache).

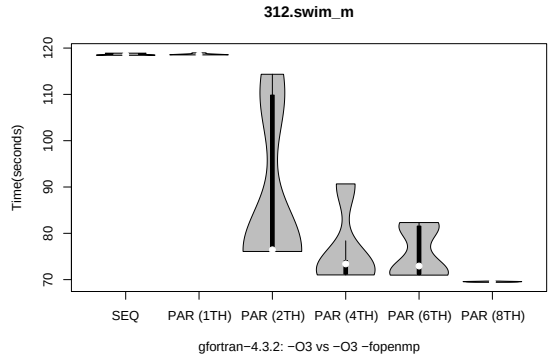
4.4 Study of the impact of thread affinity on SPEC OMP2001 execution times

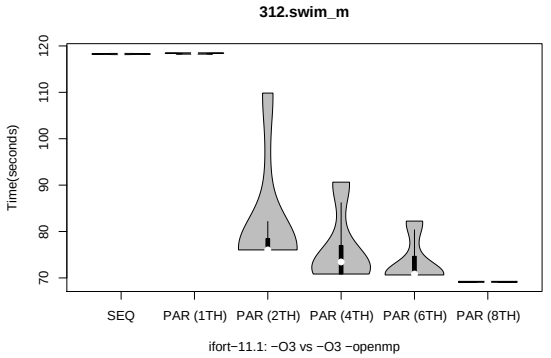
In Section 4.3, we demonstrated that contrary to sequential applications, parallel OpenMP applications suffer from a severe instability in performance. In order to analyse the reason of such performance variability, this section presents experiments aiming to measure and quantify the impact of thread affinity on the variability of program execution times of OpenMP applications. When affinity is enabled, we mean that we fix the placement of the threads on the cores of the processor. In our thesis, we restrict the number of threads to be less or equal to the number of cores.

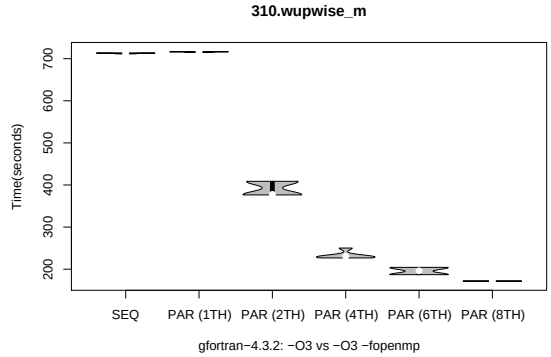
We used the `gcc 4.4.3` and `icc 11.0` compilers. For each SPEC OMP2001 application, we generated a multi-threaded version of each benchmark by setting `-O3 -fopenmp` and `-O3 -openmp` compilation flags respectively for the `gcc` and `icc` compilers. We run each application with respectively 2, 4 and 6 threads under three runtime configurations:

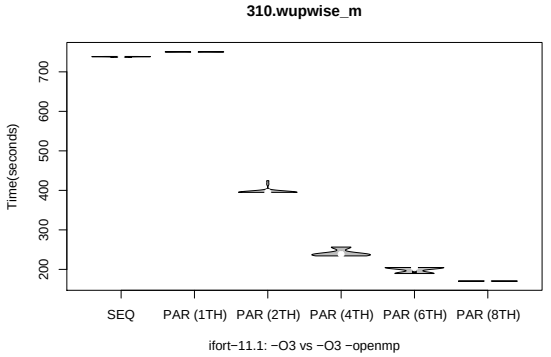
1. Running the benchmarks without scheduling affinity (affinity disabled, threads placement let to the OS).

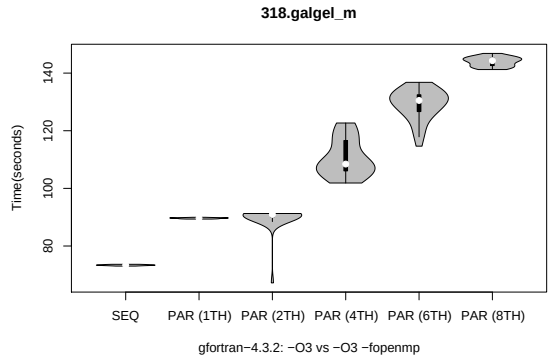
2. Running the benchmarks under the `icc` compiler `compact` affinity strategy. Specifying `compact` as affinity strategy assigns the OpenMP thread $n + 1$ to a free core as close as possible to the core where the OpenMP thread n was placed. We experiment this affinity strategy because it leads to increase the L2 cache sharing between threads, even if not all the applications can take advantage from it.











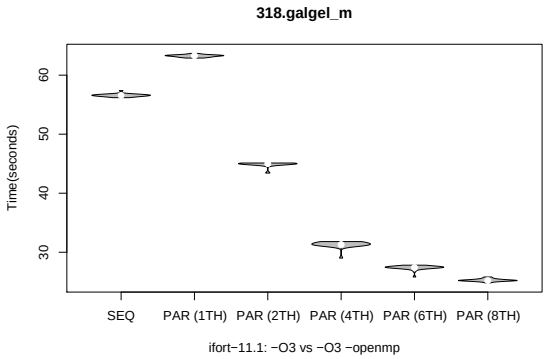


Figure 4.5: Observed Execution Times of some SPEC OMP 2001 Applications (compiled with `gcc`)

Figure 4.6: Observed Execution Times of some SPEC OMP 2001 Applications (compiled with `icc`)

3. Running the benchmarks under the `icc` compiler `scatter` strategy. Specifying `scatter` as affinity strategy distributes the threads as evenly as possible across all the sockets. `scatter` is an opposite affinity strategy compared to `compact`. Running applications under this strategy may be beneficial to alleviate the problem of system bus contention of neighbours cores.

Figure 4.12 and Figure 4.13 show violin plots of program execution times (CPU time) for the `wupwise` and `swim` applications (from SPEC OMP2001 benchmarks) compiled with the `gcc` and `icc` compilers. In each figure, three violin plots report the execution times when the benchmarks are launched with 2, 4 and 6 threads. The X-axis represents the three affinity configurations (`no affinity`, `compact`, `scatter`). The Y-axis represents the 31 observed execution times for each configuration. We make the following observations:

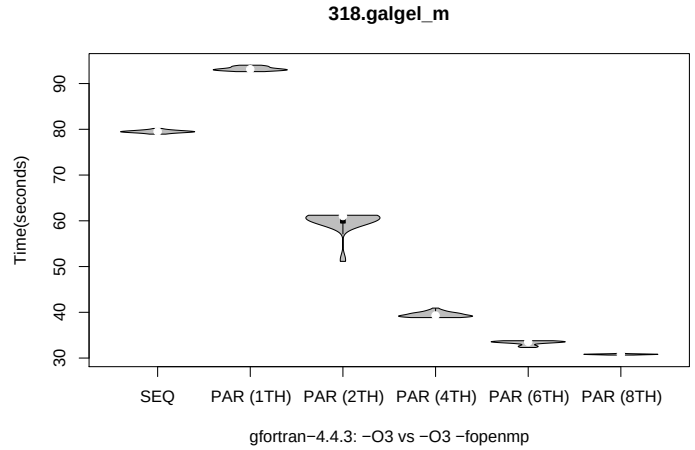


Figure 4.7: Observed Execution Times of the `galgel` benchmark compiled with `gcc-4.4.3`

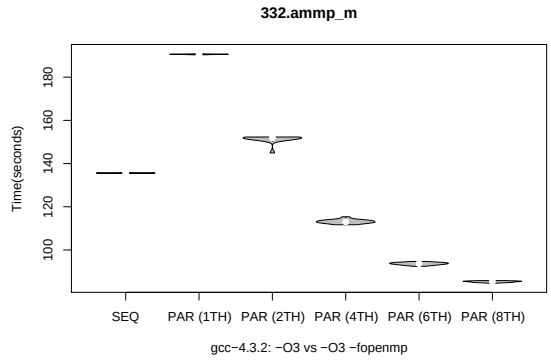


Figure 4.8: Observed Execution Times of `ammp_m` benchmark (compiled with `gcc`)

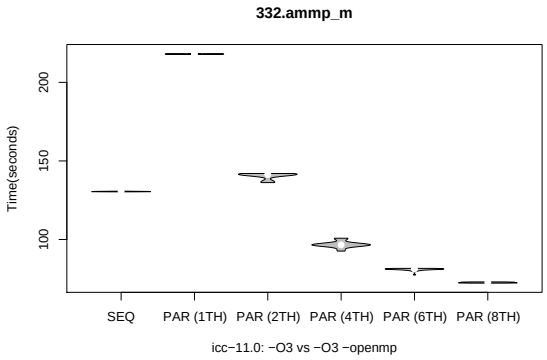


Figure 4.9: Observed Execution Times of `ammp_m` benchmark (compiled with `icc`)

1. When the scheduling affinity is disabled, we observe a significant variability of execution times for SPEC OMP2001 benchmarks. If we consider the case of `swim` in Figure 4.13 compiled with `gcc`, the version with 2 threads runs between 79 and 110 s, the version with 4 threads runs between 73 and 90 s and the version with 6 threads runs between 71 and 82 s. Figure 4.13 shows that when the benchmark is compiled with the `icc` compiler, it exhibits a variability too.

2. The variability is insignificant in almost all the benchmarks when the scheduling affinity is enabled (the observed relative variability (RV) is less than 1.5%). The variability disappears either when the threads shares L2 cache (`compact` binding) or not (`scatter` binding). Figure 4.12 shows for the `wupwise` application compiled with `gcc` that the version with 2 threads runs $\approx$ 454 s when they share the L2 cache (2 threads runs on 2 cores sharing single L2 cache `compact`) and runs between 419 and 421 s when they do not share it (`scatter`).

3. The `art` (compiled with both compilers) and the `apsi` (compiled with `gcc`) benchmarks exhibit a less sensitivity to changing scheduling affinity. We observed that even when we set up the binding feature, variability in execution times still appear (see Figures 4.14 and 4.15 where the variability exceeds 5%). In other words, fixing the affinity between

the threads does not remove the performance variability of all the benchmarks.

4. We observed in 7 out of the 9 tested benchmarks, that they run faster when they are launched with a `scatter` strategy). The benchmarks which take benefits from L2 cache sharing `compact` are `ammp` and `galgel` with both compilers (see Figures 4.16 and 4.17).

In order to check the origin of the performance variability observed when we disable the affinity, we study the impact of thread placements (fixed by the OS) on the cache effects. For instance, we run `swim` and we report its number of last level cache misses (L2 cache misses). Figure 4.11 shows violin plots summarising the number of L2 cache misses when `swim` runs with 2 threads. We observe clearly that the variability of the execution times observed in Figure 4.10 is closely related to the number of L2 cache misses. Indeed, when we binded the threads of `swim` explicitly to the system cores, we observed insignificant variability in the execution times. But when letting the system to handle threads placement on the cores, the situation was completely different and we observed an important variability. The interesting thing is that higher execution time in the configuration without affinity was accompanied with a higher L2 cache misses number. This situation shows that `swim` is sensitive to cache affinity.

When affinity is not fixed, the increase in the number of L2 cache misses does not explain the cause of the observed performance variability, but just an effect. A further analysis showed that thread migration operated by OS kernel is another important factor contributing to performance variability. Indeed, we traced the mapping of threads to cores each time a new parallel region is entered. The analysis of the tracing of the mapping event allowed us to see that the runs with high execution times, the application threads have suffered from a thread migration. Thus, migration has a negative impact on cache utilisation which leads to a significant performance variability. However, it is possible that thread migration improves execution times : this is the case for instance when data reuse and L2 cache sharing are less important.

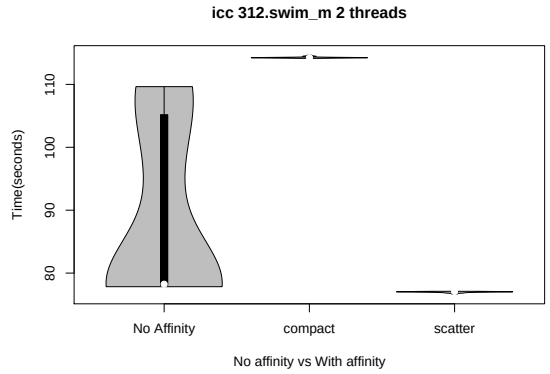


Figure 4.10: Observed cycles count in `swim` running with 2 threads

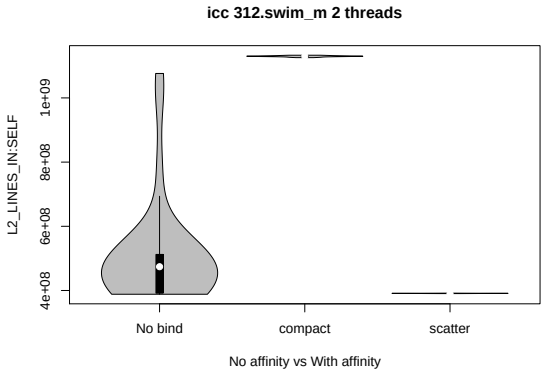


Figure 4.11: Observed L2 cache lines misses in `swim` running with 2 threads

In addition to `swim`, we observed also that the performance of `wupwise, applu, equake, apsi, fma3d, ammp` applications are sensitive to cache affinity too.

In this section, we clearly observe that fixing affinity between threads removes performance variability in many applications, but not all: there are still other influencing factors that make executions time to vary (threads synchronisation, cache access contention between threads,

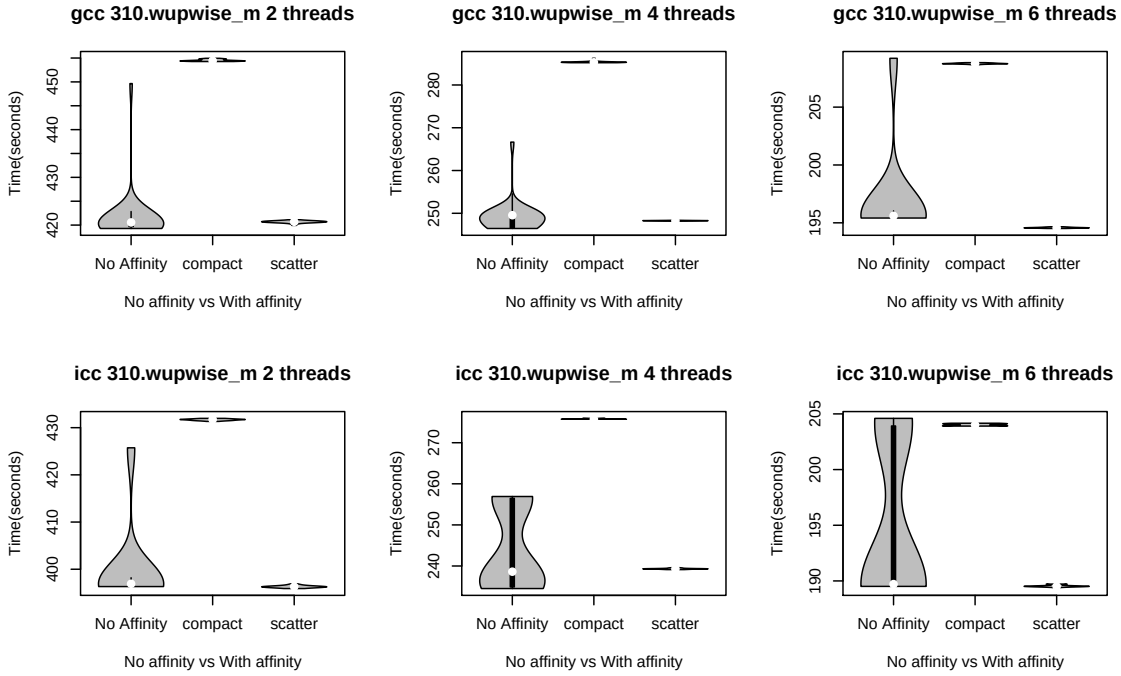


Figure 4.12: Observed Execution Times of the `wupwise` Application (compiled with `gcc` and `icc`)

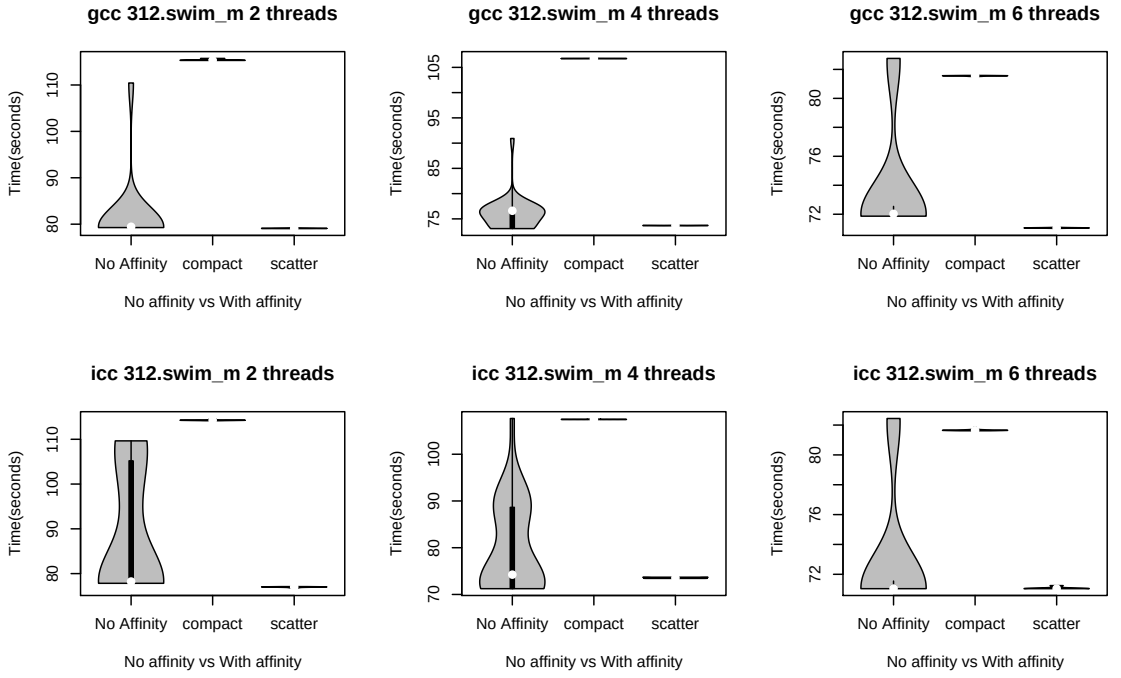


Figure 4.13: Observed Execution Times of the `swim` Application (compiled with `gcc` and `icc`)

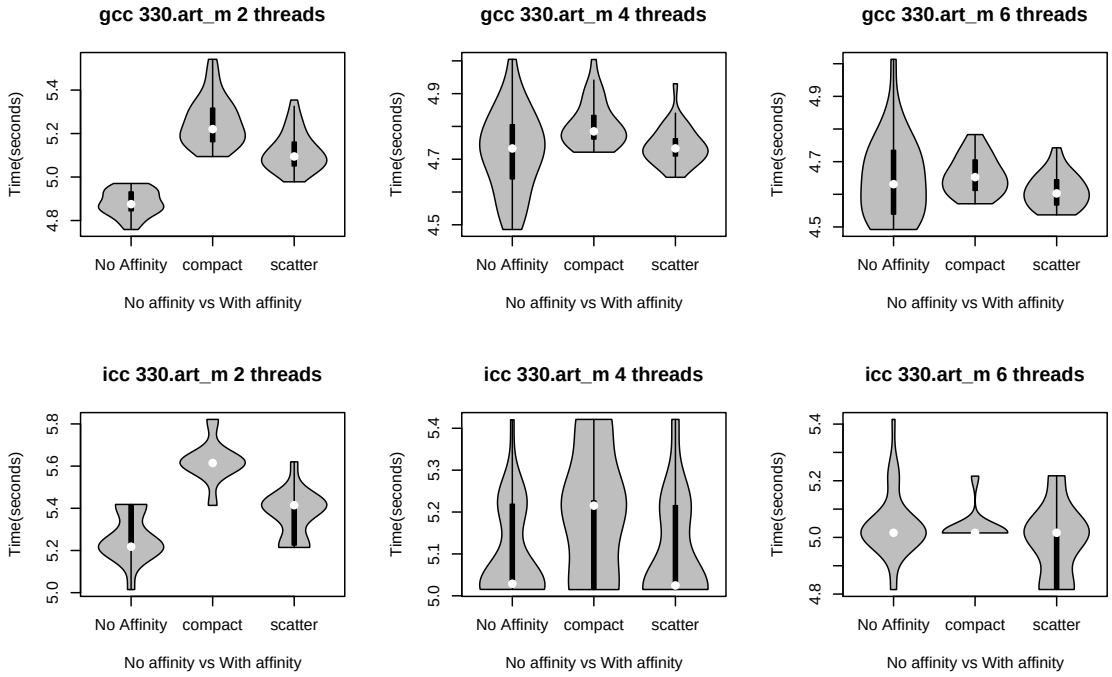


Figure 4.14: Observed Execution Times of the `art` Application (compiled with `gcc` and `icc`)

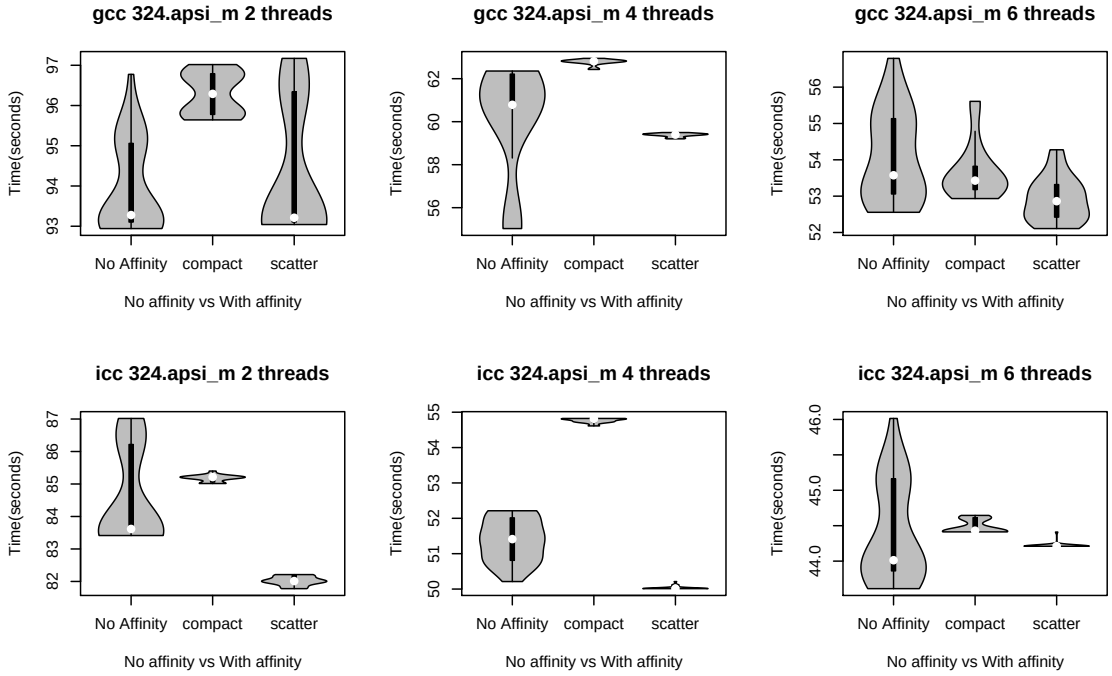


Figure 4.15: Observed Execution Times of the `apsi` Application (compiled with `gcc` and `icc`)

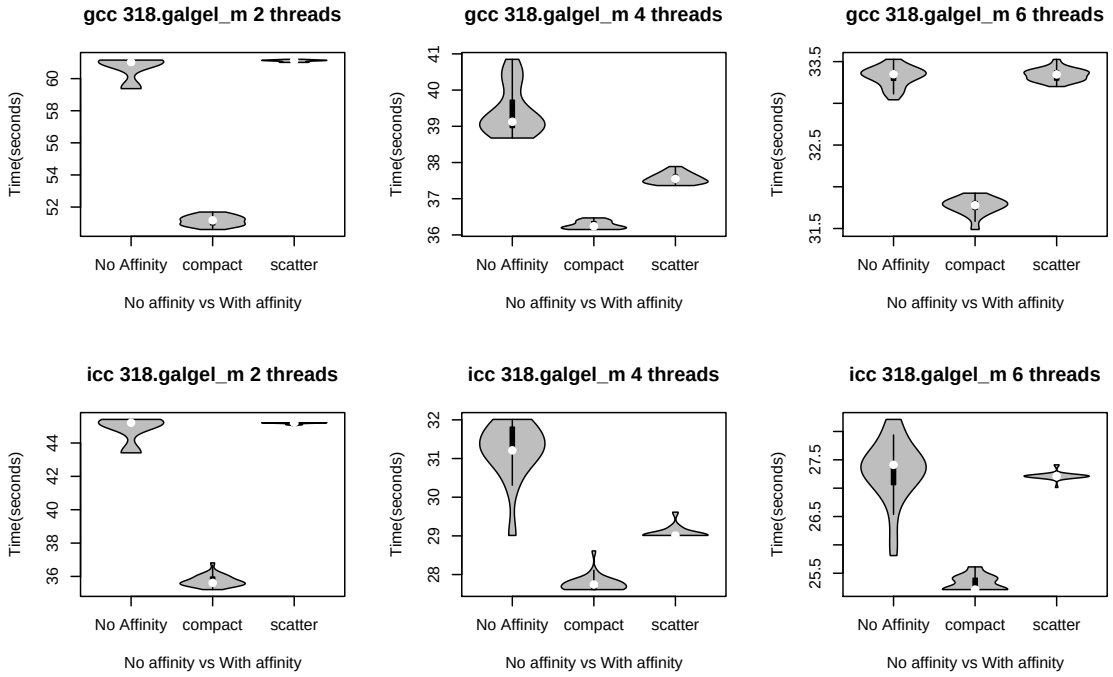


Figure 4.16: Observed Execution Times of the `galgel` Application (compiled with `gcc` and `icc`)

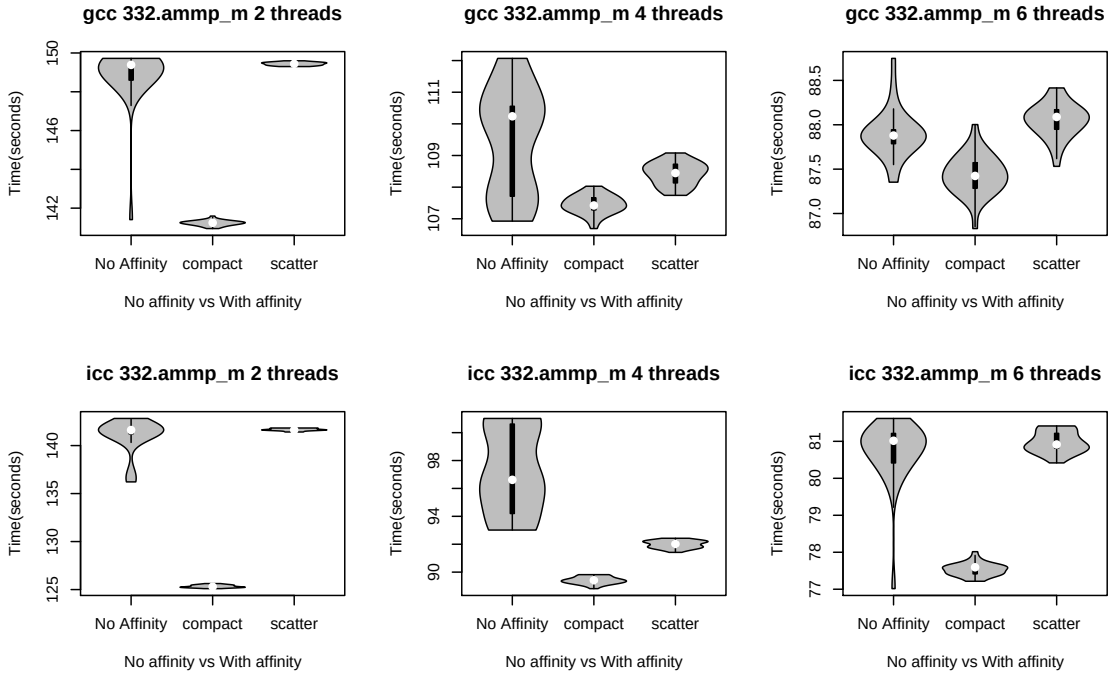


Figure 4.17: Observed Execution Times of the `ammp` Application (compiled with `gcc` and `icc`)

etc.). By now, while it is clear that thread affinity helps to reduce performance instability, it is unclear if all thread affinity strategies would be beneficial for the program performance. Indeed, we observed that 7 out of 9 benchmarks (Figures 4.12, 4.13, 4.14 and 4.15) run faster with the `scatter` strategy. This is does not mean that the `scatter` strategy would be better for all the benchmarks. Sometimes, it is also better to let some hazard (OS) to decide about thread binding. If we consider the median execution time of the `wupwise` benchmark compiled with `icc` and running with 4 threads in Figure 4.12, we can see that `no affinity` produces better performance than the `scatter` and `compact` strategies. For a further analysis of the impact of thread affinity on program performance, we study in Chapter 5 the performance of OpenMP programs under various affinity strategies. The next section explores the performance variability when SPEC OMP01 are executed in parallel with other co-running processes.

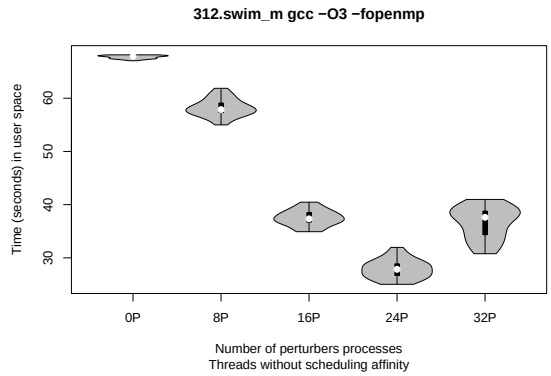
4.5 Analysing the variability of SPEC OMP performance with co-running processes

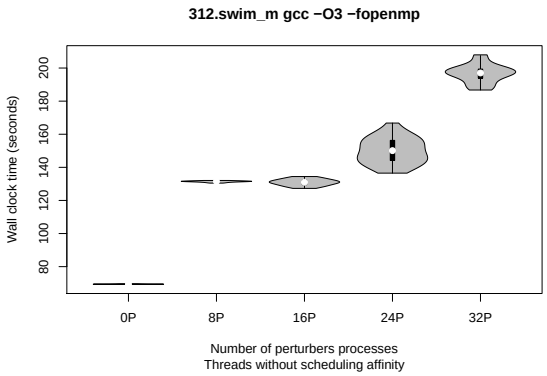
One of the factors which can influence the variability of program execution times is when more than one thread is scheduled to run on top of a single core. In our study, we focus on the sharing between the OpenMP parallel programs and some artificial concurrent applications. For each OpenMP benchmark, we measure its execution times in user mode (run level 3 : least privileged mode), system mode (run level 0 : most privileged mode) and real execution time (total elapsed execution time).

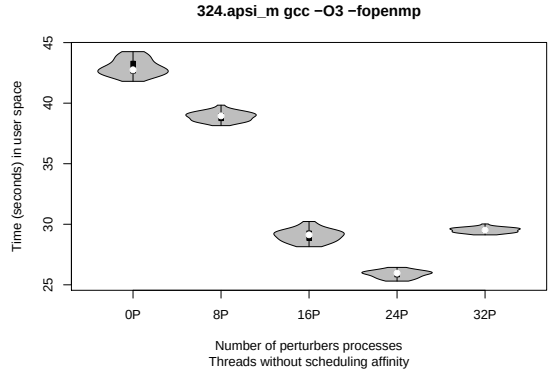
In these experiments we generate a system load by running some artificial co-running processes in background. These processes are launched by one process executing the `fork` system call a number of times equal to the number of processes that we need to generate at runtime. This number is supplied as an argument to the command line. The code executed by these co-running processes is a dummy non terminating loop without memory access (**do while(1)**;). Note that the threads of an OpenMP benchmark and the co-running processes do not share the same internal memory, they are independent applications.

4.5.1 Experimental setup

- The SPEC OMP2001 benchmarks are launched with 8 threads at runtime to occupy all the cores of the system.
- Each SPEC OMP2001 benchmark (8 threads) is run either as a single application on the machine (minimal system load) or in parallel with 8, 16, 24 or 32 co-running processes respectively (performance perturbation created in background). This leads to five distinct runtime configurations.
- We report here the results when SPEC OMP2001 and the co-running processes are launched without scheduling affinity to the system cores (no explicit binding of threads on cores). Similar experiences have been conducted when affinity is fixed, the conclusions remain similar.
- The number of the OpenMP threads (from applications under study) and co-running processes running on each core are respectively: 1, 2, 3, 4 and 5. For example, a configuration with 5 threads or co-running process per core consists of 1 OpenMP thread plus 4 co-running processes.







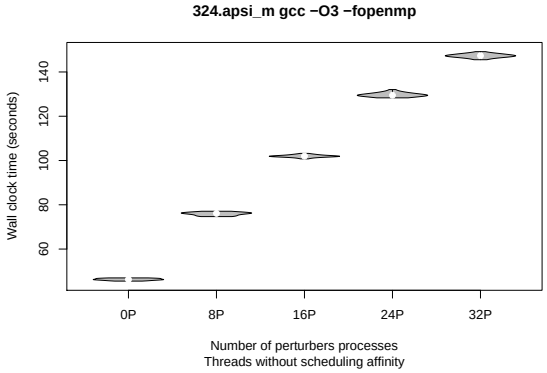


Figure 4.18: Observed User Execution Times of some SPEC OMP2001 Applications (compiled with `gcc`)

Figure 4.19: Observed Real Execution Times of some SPEC OMP2001 Applications (compiled with `gcc`)

4.5.2 SPEC OMP2001 with co-running processes performance results and analysis

Figure 4.18 and Figure 4.19 show the violin plots of the user and real program execution times for four applications from SPEC OMP2001 benchmarks compiled using the `gcc-4.3.2` compiler. The X-axis represents the violin plots of program execution times when the SPEC OMP2001 benchmarks run together with the co-running processes. The Y-axis represents the 31 observed execution times for each software configuration. In each violin plot, we still have non-negligible performance variability since thread affinity is not fixed.

In addition, a strange phenomenon appears. From Figure 4.18, we can see that when we increase the number of processes running in background, the program execution times at user level of the OpenMP applications decreases. Meanwhile, we observe in Figure 4.19 an increase of real execution times as expected. Running the threads of the SPEC OMP2001 benchmarks and the co-running processes with a fixed scheduling affinity leads to the same conclusion (not plotted here): when we increase the number of co-running processes, we observe a decrease in program execution times at user level of the OpenMP applications. There are multiple factors that may explain such phenomena. These factors can be classified mainly into micro-architectural or OS interactions. In order to make an analysis of this phenomena, we design micro-benchmarks (SPEC OMP2001 are too complex to analyse directly) to isolate some micro-architectural and OS events for this application behaviour. The next section describes our study with micro-benchmarks.

4.6 Analysing the variability of micro-benchmarks with co-running processes

In Section 4.5, we performed experiments where SPEC OMP01 applications run simultaneously with co-running processes. We showed that increasing the number of co-running processes may decrease the execution time at the user level. In order to understand these observations, we used synthetic micro-benchmarks. The idea behind such benchmarks is to isolate micro-architectural, application and operating system level events that may have an influence on the observed performance behaviour. To do so, we use two classes of micro-benchmarks: 1) memory-bound benchmarks (make intensive access to the memory), and 2) CPU-bound benchmarks (do not make intensive access to memory).

4.6.1 Memory-bound micro-benchmarks

We start our evaluation by memory-bound benchmarks. This class of micro-benchmarks stresses the memory cache hierarchy in order to understand the relation between increasing the number of co-running processes and the decreasing of the user execution times.

Micro-benchmarks code

The code of the micro-benchmarks in Listing 4.1 is composed of three loops `Loop1` (parallel loop), `Loop2`, `Loop3` and one statement `S1`. The `L2` loop is added for repetition purpose to increase the measurement accuracy. The data set accessed by all the micro-benchmarks is equal to $\mathsf{N} * \mathsf{M} * sizeof(int64) = D\ bytes$ where N is the number of iterations of the outermost loop (`Loop1` loop) and M is the number of iterations of the inner most loop (`Loop3` loop). In addition, since we want to give the same workload to every thread, this leads to consider values for N which are multiple of the number of threads. Having all these constraints, the values taken by N are from 8 to 196608 and the values taken by M are from 196608 to 8. For instance, when we have 8 threads, the value of N starts at 8. Furthermore, whatever the values of N and M are, the workload assigned to each thread has a working set of size $1.5MB$. This size is chosen to be less than the half of the size of the L2 cache preventing from frequently accessing the DRAM in case of L2 cache misses.

Now we define the notion of memory *chunk*. In the context of our study the *chunk* represents the size of the vector fraction from `tab` accessed by the innermost `M`-loop (loops): in the particular code of Listing 4.1, we clearly see that each iteration of the `M`-loop (loops) accesses to a single element from `tab`, consequently $chunk\ size = \mathsf{M} \times sizeof(int64)$. Table 4.1 gives the couples of values of N and M that we have experimented. To every couple of values (N,M) we associate the micro-benchmark mb_N-value_M-value. Following this denomination, the first micro-benchmark is mb_8_196608, the second mb_16_98304 and so on. With this micro-benchmark structure, we access the array `tab` in an indexed way represented by the `S1` statement. Thus, the size of the chunk accessed in the `M-loop` depends on the number of iterations of the `Loop1`. So, larger values of N lead to smaller sizes of chunks. In contrary, smaller values of N lead to larger sizes of chunks.

N	M	Benchmark	Chunk size = M*sizeof(int64)	N	M	Benchmark	Chunk size = M*sizeof(int64)
8	196608	mb_8_196608	1536 KB	1536	1024	mb_1536_1024	8 KB
16	98304	mb_16_98304	768 KB	3072	512	mb_3072_512	4 KB
32	49152	mb_32_49152	384 KB	6144	256	mb_6144_256	2 KB
64	24576	mb_64_24576	192 KB	12288	128	mb_12288_128	1 KB
128	12288	mb_128_12288	96 KB	24576	64	mb_24576_64	0.5 KB
256	6144	mb_256_6144	48 KB	49152	32	mb_49152_32	0.25 KB
512	3072	mb_512_3072	24 KB	98304	16	mb_98304_16	0.125 KB
1024	1536	mb_1024_1536	12 KB	196608	8	mb_196608_8	0.0625 KB

Table 4.1: Values taken by N and M in the outermost and the innermost loops of th micro-benchmarks code

Listing 4.1: OpenMP micro-benchmarks code

```
void wastetime() {
  #pragma omp parallel for default(none) private(i,j,k) shared(tab)
Loop1:   for(i = 0 ; i < N ; i++)
Loop2:         for(j = 0 ; j < 10000; j++)
Loop3:               for(k = 0 ;  k < M; k++)
S1:                    tab[i*M+k]++;
}
```

The first micro-benchmark mb_8_196608 corresponds to the case where every thread executes a single outer loop iteration (`Loop1` or the `N-loop`) and the innermost loop `M-loop` accesses to a chunk of size $M * sizeof(int64) = 1.5\ MB$. This chuck size is sufficient to keep data inside the L2 cache but not inside the L1 data cache. In other words, the first micro-benchmark guarantees that every thread has its data in L2 but not in L1. The last micro-benchmark mb_196608_8 corresponds to the case where every thread executes $N/8 = 24576$ `N` iterations, the innermost `M`-loops access to a chunk of size $M * sizeof(long) = 64\ B$ (a single cache line size). In other words, this last micro-benchmark guarantees that every thread has all its data in L1. The other micro-benchmarks between mb_8_196608 and mb_196608_8 cover the range for other values of (N,M). They give us the performance of the intermediate situations when data are fully or partly in L1. We should have (in theory) all data fully inside L2 because the chunk sizes are all less than half of L2 size, but we see later that threads sharing common L2 may create cache conflicts, thus data are ejected from L2.

Testing different co-running processes

We investigated three types of co-running processes:

1. CPU-bound co-running processes which are simple dummy non terminating loops (`do while (1);`).

2. CPU-bound with sleeping state: that is, the co-running process makes a call to the `usleep` function with various values (from 10 μs to 10 ms, this later value is the default Linux kernel time slice). It allows us to study the behaviour of the micro-benchmark when its co-running processes sleep completely or partially.

3. Memory-bound co-running processes: that is, the co-running process makes extensive accesses to the L2 cache to be in competition with the micro-benchmarks. However, the working set of a co-running process does not exceed half of the size of L2 cache.

The experimental environment

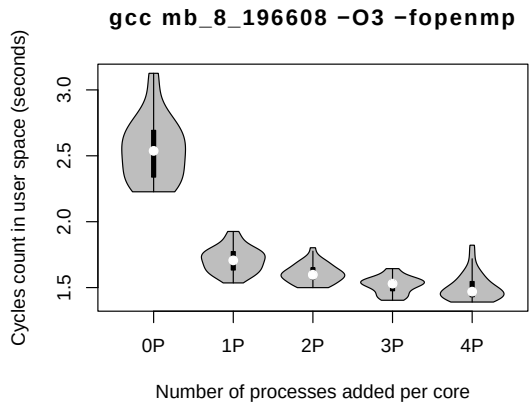
The micro-benchmarks and the co-running processes are executed concurrently under multiple software and environmental configurations:

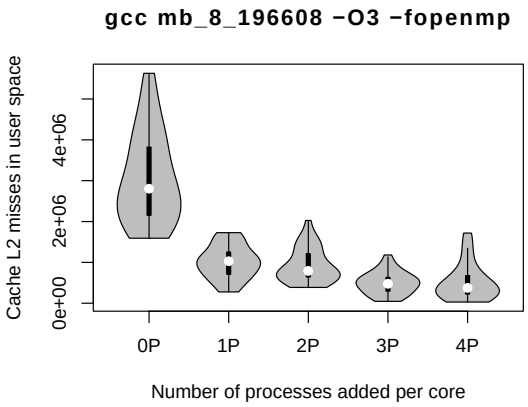
- All of the micro-benchmarks and the co-running processes were compiled with the compiler optimisation flag `-O3`.
- The number of threads of the micro-benchmarks is either 8 (to occupy all the cores) or 4 (to test some affinity configurations).
- Affinity is fixed in order to experiment two situations: sharing of L2 cache between threads or not (when 4 threads are used, sharing or not the 4 L2 caches of the system).
- The number of co-running processes per core is varied from 0 to 4.
- We tested the case of active and inactive automatic hardware prefetching.
- We tested two values of memory pages: small size (4 KB) and large size (2 MB).

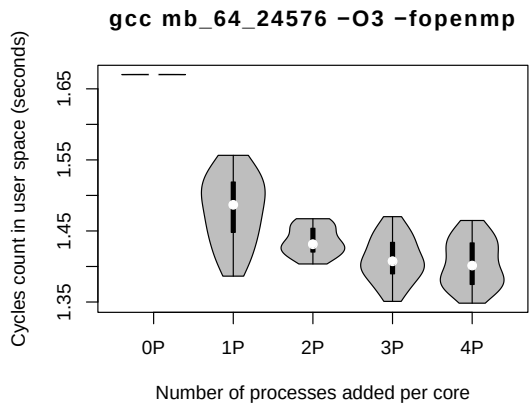
Memory-bound micro-benchmarks performance results and analysis

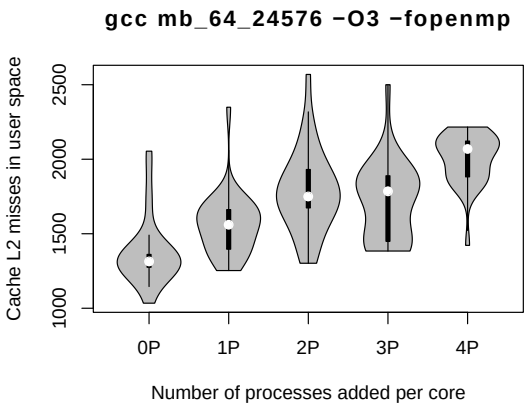
This section presents a synthesis of the experimental evaluation. The full results and analysis are given in [MTB10]. First, we confirm the observation done in Section 4.5: given a parallel OpenMP application, increasing the number of background co-running processes may decrease the execution time at the user level (see Figures 4.20 and 4.21) while the real execution time increases. Indeed, our extensive experiments with the different micro-benchmarks and co-running process described in this section run in different configurations (different affinity, page sizes, disabling/enabling automatic hardware prefetch) allow us to figure out the following observations:

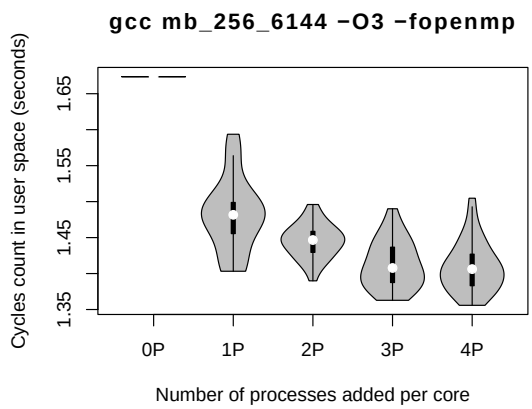
- When we use memory-bound co-running processes, we do not observe any decrease in user level program execution times, we observe an increase instead.
- When we use CPU-bound co-running processes, the decrease of execution times at user-level was observed on micro-benchmarks which access chunks having a size greater than the size of the L1 data cache. Thus, this subset of micro-benchmarks needs a high frequency access to the L2 cache to provide the needed data by the L1 data cache leading to a high number of L1 data cache misses. Moreover, this phenomena was confirmed for configurations where at least two threads are running on two adjacent cores. In this situation, the competition to access the shared L2 cache is increased. We confirmed this observation with experiments using four threads. Indeed, running four threads on an eight cores machine allowed us to test two configurations: 1) threads that do not share the L2 cache, and 2) threads that share the L2 cache.
- The automatic hardware prefetching combined with a small pages size generates L2 cache conflicts (which translates to cache misses), even if enough cache capacity exists to hold all the accessed data. Such L2 cache conflicts explain part of the performance variability observed in our experiments.

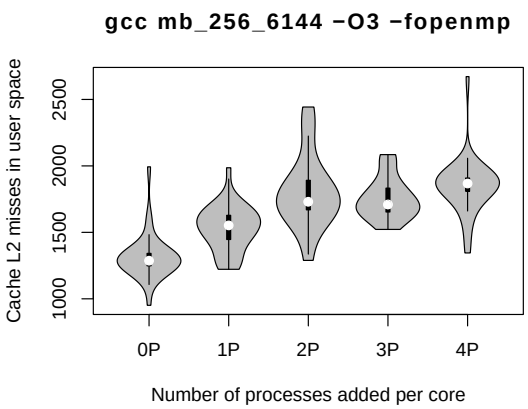


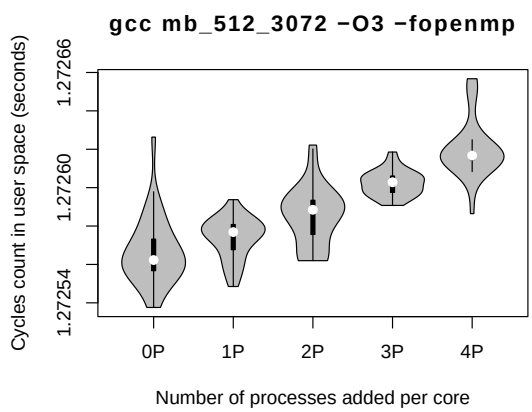












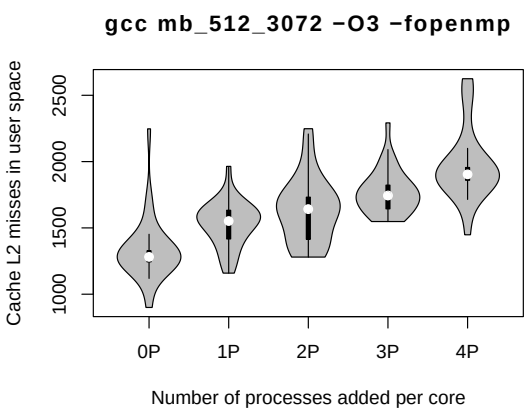


Figure 4.20: Observed User Execution Times of some micro-benchmarks (compiled with `gcc`)

Figure 4.21: Observed L2 cache misses of some micro-benchmarks (compiled with `gcc`)

In the light of these observation, we can say that co-running processes contribute to significantly reduce contention on L2, decreasing the program execution time at the user level. We confirmed this observation by disabling the automatic hardware prefetching and using large pages of 2 MB. The decrease of the execution time at the user level when we increase the number of co-running processes is due to the following fact: the co-running processes run in competition with the OpenMP threads. Consequently, the threads of the same application run with less competition between themselves, their access to L2 cache is regulated (smoothed) by the co-running processes. In other words, the contention on the L2 cache and on the memory bus is reduced thanks to the co-running process which consume fraction of the CPU time. Moreover, using co-running processes with different sleeping states confirmed our findings. While co-running processes spending long periods in the sleeping state do not decrease the user level execution times, co-running processes with short periods effectively help to decrease it.

The next section presents a similar performance study using CPU-bound micro-benchmarks.

4.6.2 CPU-bound micro-benchmarks

In the previous sections, we presented performance data related to OpenMP applications intensively accessing the memory hierarchy (particularly the L2 cache). But what is the expected behaviour when an OpenMP application has a low access rate to memory? Answering the latter question leads us to experiment running applications having less memory accesses.

Micro-benchmarks code

As CPU-bound benchmark, we used a `prime-number` benchmark. Listing 4.2 shows the code of this application. As we can see from the listing, the code is CPU-bound (the memory access is limited to L1 data and instruction caches).

Listing 4.2: OpenMP primenumber code

```
int prime_number ( int n ) {
  int i, j, prime, total = 0;

  # pragma omp parallel shared (n) private (i,j,prime) reduction (+:total)
        # pragma omp for
        for ( i = 2; i <= n; i++ ) {
        prime = 1;
        for ( j = 2; j < i; j++ ) {
            if ( i % j == 0 ) {
                prime = 0;
                break;
            }
        }
        total = total + prime;
    }
    return total;
}
```

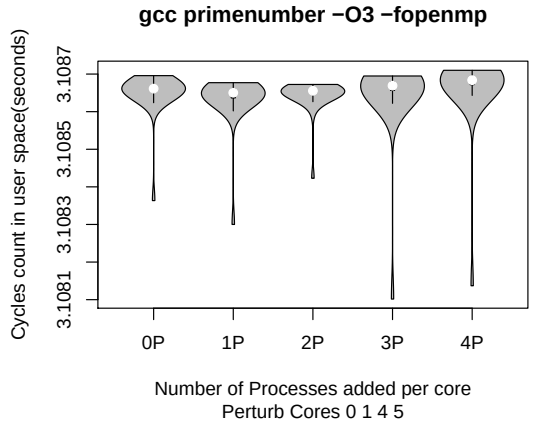


Figure 4.22: Observed User Execution Times for prime number application running on the 0, 1, 4 and 5 cores

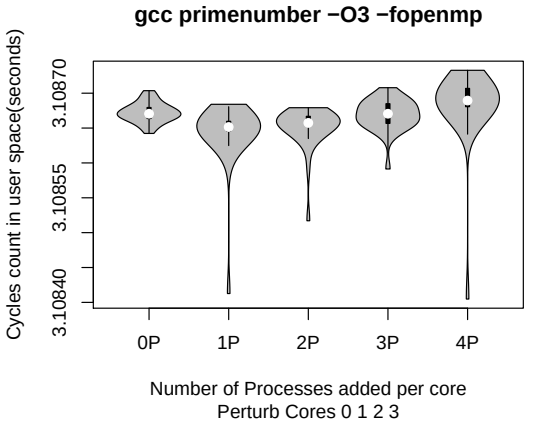


Figure 4.23: Observed User Execution Times for prime number application running on the 0, 1, 2 and 3 cores

The experimental environment

The `prime-number` application is executed with 4 OpenMP threads. The application (4 threads) is run either alone as a single application on the machine (minimal system load) or in parallel with 4, 8, 12 or 16 CPU-bound co-running processes. We have also two scheduling affinity configurations:

1. The application threads and the co-running processes are launched on the system cores 0, 1, 4 and 5, where cores 0 and 4 share one L2 cache, similarly cores 1 and 5 share another L2 cache.
2. The application threads and the co-running processes are launched on the system cores 0, 1, 2 and 3, where there is no sharing of L2 cache between these cores.

CPU-bound micro-benchmarks performance results and analysis

Figures 4.22 and 4.23 show program execution times for `prime number` application running on shared L2 caches (0,1,4,5 cores) and non shared ones (0,1,2,3 cores) respectively. The interesting observations from these experiments are:

1. Running the application on the two runtime affinities does not change user execution times.
2. The variability of user execution times for each violin plot on the figures is negligible (looking very carefully to the Y-axis).
3. Running the co-running processes affect only the whole (real) execution time. Indeed, the whole execution times increases when we add more co-running processes. This situation is expected since there are many processes in the system and they need some time to complete.

4.7 Conclusion

In this chapter, we showed clearly that, even if a machine has low overhead and the dynamic voltage scaling is inactive and the automatic hardware prefetcher is disabled, the execution

times of OpenMP applications on multicore platforms may be unstable (variable). This implies to study a new performance criteria for code optimisation that was not important for sequential codes, which is performance stability. We showed that binding threads on cores removes the performance variability in most of the cases (when the number of threads does not exceed the number of cores), but some applications still have unstable program performance after fixing the thread affinity. This means that other factors (distinct from thread binding) are still playing important role on performance variation.

Our study also highlights that executing separate co-running processes in parallel with the threads of an OpenMP application may be beneficial for the user level execution time (but not for the whole real time execution). Indeed, co-running processes may reduce the contention or competition between the threads on data that reside on shared cache levels: co-running processes push the threads of the OpenMP application to run with less concurrency, smoothing the conflicts on shared cache levels. While co-running processes are not beneficial for real execution times, they contribute to reduce the user level execution times, which means that the efficiency of the whole system is improved (fraction of time where the CPU really executes applications is improved).

The speedups that are reported in the literature are usually observed in ideal environments, in ideal experimental setups, after retaining *good* execution times. The end-user however may not observe the declared speedups, which may cause frustration. The reason is that end users do not work in ideal environments: they may not know what are the hidden factors that influence the performance stability of their codes, or simply they may not have a root privilege (or enough rights) to the machine to fix it. Consequently, when an end-user executes an application that is declared optimised, he would have a low chance to observe such performance improvements with the declared speedup.

The next chapter studies the performance benefit and performance variability of multiple data sharing aware thread binding strategies.

Chapter 5

Thread Affinity Techniques for OpenMP Applications on Multicores

This chapter presents a study on the performance of various thread affinity strategies. It investigates the performance of *application independent* strategies (heuristics) and *application dependent* strategies. The later strategies are computed using a profile guided method based on the amount of data sharing exhibited by multi-threaded applications.

5.1 Introduction

Multicore processors do not change fundamentally parallel computing paradigms. Any parallel application can be run safely on multicore processors as if it is run on a classical multi-processor machine. The operating system (OS) considers every core as a distinct processor. If we have a processor with, say 8 cores, the OS sees 8 homogeneous processors that are capable of executing concurrent threads, processes or jobs. However, in terms of performance tuning, we cannot consider the cores as homogeneous because they share common micro-architectural resources: L2 or L3 shared caches, shared memory buses, etc. Consequently, the placement of threads on the cores, called *thread pinning*, is of high importance. For instance, if two threads make extensive accesses to common data in memory, it is better to place them on adjacent cores sharing the same L2 or L3 cache, or the same NUMA node. Data locality and reuse are not the unique performance factors that influence program execution times in case of concurrent applications. Other factors have an influence: memory bus bandwidth, non uniform memory access (NUMA) effects, OS (synchronisation costs, Input/Output, thread scheduling), etc. In this chapter we focus on the study of cache sharing effects.

In Chapter 4 we studied performance variations when we execute an OpenMP application multiple times (with the same data input) in a batch mode. We demonstrated the following conclusions:

- Work balancing is satisfactory in terms of core usage ratio because threads are scheduled and placed to optimise the usage of all cores.
- In terms of performance, execution times exhibit high variations. For every new run of the parallel application (with the same data input), the OS may decide for a different thread placement. In addition, threads may be migrated from one core to another to improve work balancing and core utilisation ratio.

- When thread affinity is fixed, performance variations are greatly reduced, but are still present in few cases.

- From the performance stability perspective, compared to memory page size or hardware prefetching, thread affinity is a dominating factor in multi-threaded applications.

Work balancing is a classical performance criteria in parallelism and task scheduling, targeted by OS, distributed systems, grid computing, etc. However, its direct impact on code performance is not guaranteed. The reason is that work balancing aims to keep all cores busy. While all cores may be executing threads, performance may still be low because of poor interplay between the code and the micro-architecture: threads can spend most of their times servicing cache misses, doing pipeline stalls and branch mispredictions. Consequently, work balancing may improve synchronisation costs by making all threads reach the synchronisation barrier jointly, but with poor performance.

Thread affinity has quickly appeared to be one of the most important factors that impact program execution times on multicore processors. Still now, it is not clear how to decide for the best thread placement that considers all the possible performance factors (data locality, memory bus bandwidth, OS synchronisation overhead, NUMA effects, etc.). In this chapter, we present an empirical study of nine thread placement strategies on three distinct machines. Among them, four strategies are application independent: they apply the same thread placement decision whatever the application. Five strategies are dependent on the application: they fix the thread placement after a profile-guided analysis. For popularity in the HPC community and availability for our work, the three test machines are based on Intel X86 (64 bits) with three distinct designs. Other multicore architectures may be tested with exactly the same methodology.

This chapter is organised as follows. Section 5.2 describes all the tested thread placements techniques (called also thread pinning). Section 5.3 describes our experimental setup and methodology. The results of our experiments are detailed and analysed in Section 5.4, then we conclude. For the rest of this thesis, we use the terms threads pinning, threads affinity and threads placement without distinction. They all define the used core for every running thread.

5.2 Tested thread pinning techniques

We experimented various thread affinity techniques. We classify these techniques into two main families: 1) *application independent* and 2) *application dependent* thread pinning strategies. The former family places threads on cores independently of the characteristics of the program; for any application, threads are placed in the same way. The later places threads on cores according to the characteristics of the program. In our study, the characteristics of the program are computed using a profiling phase. We focus on data sharing to decide about the best thread placement.

5.2.1 Application independent thread pinning techniques

The application independent class consists of the following four thread affinity techniques:

1. Run the application without affinity (`no affinity`), this means that we let the OS decide about the thread placement on the cores of the machine. This strategy allows thread migration between cores during the execution of the application.

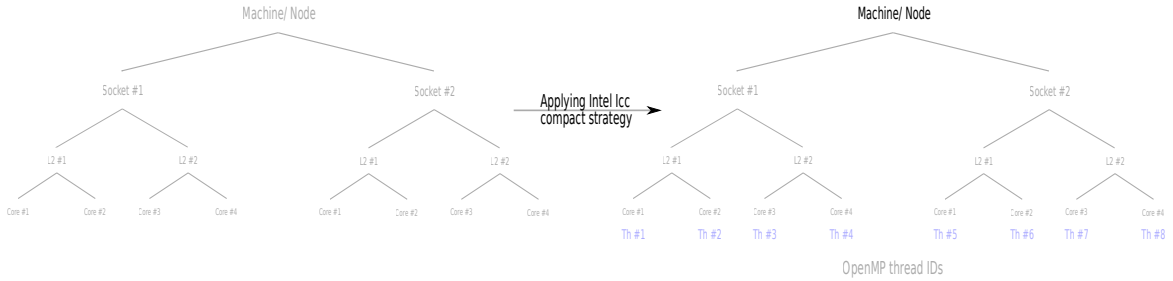


Figure 5.1: Thread placement following the `icc compact` strategy on a machine with two sockets, each socket has four cores where each pair of cores share an L2 cache

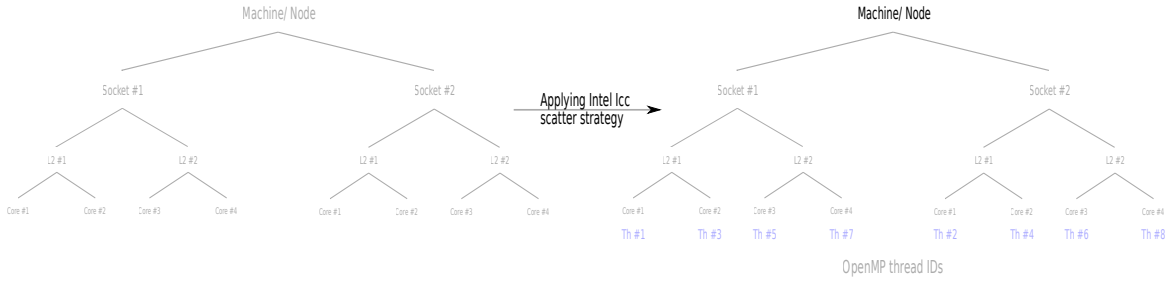


Figure 5.2: Thread placement following the `icc scatter` strategy on a machine with two sockets, each socket has four cores where each pair of cores share an L2 cache

2. Run the application with a set of affinities generated randomly (`random`). Each repetitive run (35 runs) corresponds to a new random affinity. The randomly generated affinity is fixed during the whole run, there is no thread migration during a single run.

3. Run the application with a `compact` (Figure 5.1) strategy of the `icc` compiler (`icc compact`). This strategy assigns successive (in order of their creation) OpenMP threads to cores as close as possible in the topology map of the platform. This strategy is convenient for applications that have a high data reuse between threads, so they can profit from shared caches inside sockets.

4. Run the application with a `scatter` (Figure 5.2) strategy of the `icc` compiler (`icc scatter`). This strategy distributes the OpenMP threads as evenly as possible across the entire sockets. This strategy is convenient for applications that have a high data locality inside each thread, so they can profit from a large private cache inside a socket without sharing it with other threads.

5.2.2 Application dependent thread pinning techniques

This family of thread pinning techniques relies on a profile-guided method. The collected profiles are used to exploit and to maximise the opportunities of data sharing and data reuse between threads running on adjacent cores sharing common cache levels (L2 or L3).

The profile-guided approach is divided into four main steps:

1. Performing a memory tracing of the multi-threaded application.
2. Building an *affinity graph* describing data cache line sharing between every pair of threads.
3. Computing a thread pinning based on the affinity graph.
4. Running the application using the computed affinity.

Below we detail these steps.

5.2.2.1 Step 1: memory trace profile collection and analysis

We used `Pin` [LCM+05] to achieve the memory tracing of the parallel multi-threaded applications. `Pin` is a tool for dynamic binary instrumentation of programs allowing arbitrary code (written in C or C++) to be injected at arbitrary places in the executable. `Pin` does not instrument an executable statically by rewriting it, but rather adds the code dynamically while the executable is running. This also makes it possible to attach `Pin` to an already running process. `Pin` provides also a rich API that allows context information such as register contents to be passed to the injected code as parameters or collecting memory references. `Pin` automatically saves and restores the registers that are overwritten by the injected code so the application continues to work.

We implemented a C++ Pin tool using the API provided by `Pin`. Our plug-in allows us to detect the creation of each thread in the application. It allows us also to monitor all the memory instructions of these threads. The detailed functionality of the tool is explained bellow.

To be able to collect all the memory references for each thread, we instruct `Pin` to insert two instrumentation routines in the application code. The first routine registers a notification function that is called when a new thread starts executing in the application. The call-back happens even for the application's root (master) thread. The second instrumentation routine tells `Pin` to add a function which is used to instrument at the instruction granularity. When this function is called, it performs a test to check whether it is a memory read or a memory write instruction. Depending on the kind of the memory operation, the instrumentation routine sets a call-back to two analysis routines `memRead` for read operations and `memWrite` for write operations. Both analysis routines are called with the following arguments: the instruction pointer, the virtual address of the referenced memory, the size of the referenced memory and the thread identifier to distinguish between the memory references of all the threads.

We associate to each new thread one hash table. Therefore, for an application running with n threads , we create n hash tables. Each hash table holds all the memory references of a given thread. The entries of a hash table store the block identifier (BID) of the given memory reference and the number of accesses to this BID. In reality, we distinguish between the number of memory reads and the number of memory writes. The BID represents a data block of 64 bytes size (a cache line granularity). We have to notice that we store only one instance of the same BID per thread. Thus, if a thread performs multiple memory references belonging to the same BID, the hash table holds only one entry for this BID and the number of reads and writes is updated accordingly. Furthermore, for each hash table, we can deduce the total number of BID (cache lines) accessed and the total number of all the memory accesses.

We have to notice that when an application is analysed using our Pin tool, all the hash tables (one per thread that holds the accessed memory references) are only stored in memory. This

means that we do not dump the memory access trace to disk, all computations are performed in memory. Once the application has finish its execution (but before the return from Pin), we compute the inter-thread data sharing information. Having the later information, we print a final application report profile, giving the useful memory access information to build the *affinity graph* presented in the next section.

5.2.2.2 Step 2: affinity graph model

The collected memory trace profile is used to build an *affinity graph* for each application. It is an undirected valued graph $G = (\mathcal{V}, \mathcal{E}, \alpha)$. $\mathcal{V}$ is the set of application threads, $\mathcal{E} = \mathcal{V} \times \mathcal{V}$ and $\alpha : \mathcal{E} \mapsto \mathbb{N}$ is a gain function applied to every pair of threads. An affinity graph is a complete graph: we consider a fixed number of threads equal to $n = \|\mathcal{V}\|$, then the number of edges in the graph is equal to $\frac{n \times (n-1)}{2}$.

The gain function models the attraction factor between each pair of threads. For instance, since we rely on data reuse between threads to compute an affinity, the gain function $\alpha(T_i, T_j)$ represents the number of common accesses to common memory caches lines, accessed by both the T_i and T_j threads. Let us precisely define α for an application with a fixed number of threads $n = \|\mathcal{V}\|$. The collected memory trace profile contains the information $A(T_i, b)$ which is the number of accesses of thread T_i to data block b. Let $B_{i,j}$ be the set of all data blocks accessed by the pair of thread (T_i, T_j). Equation 5.1 defines the function $\alpha(T_i, T_j)$, which is exactly the number of accesses to common memory blocks by both the threads T_i and T_j (with $T_i \neq T_j$):

$$\alpha(T_i, T_j) = \sum_{b \in B_{i,j}} \min\left(A(T_i, b), A(T_j, b)\right) \tag{5.1}$$

One may wonder why we do not use a reuse distance analysis to compute effective thread affinity. As we showed in Chapter 2 Section 3.2, a reuse distance analysis is used to approximate the cache miss rate of an application. For a multi-threaded application, a reuse distance analysis can be computed in two ways. First, it can be computed with a single reuse distance profile for all threads where memory references of all threads are processed by that reuse distance. By doing so, the computed reuse distance studies the performance of a global and unique shared cache. This situation is not realistic in our case since we target machines with multiple caches. Second, it can be computed with a distinct reuse distance for each thread, and merge individual reuse distances. The main problem with this approach is related to the way the merge is performed. Indeed, should the merge model test all the sharing configurations between threads? The merge model may be impractical for large number of threads. Moreover, deeper cache hierarchies and larger data inputs can exacerbate this problem. Besides, a merge model has also the drawback to compute a single data reuse profile. It means again that the computed reuse distance targets a single shared cache. For this reason, we think that using reuse distance analysis is inadequate for our purpose to compute effective thread affinities.

The next section shows how to use the affinity graph to compute an affinity between threads exploiting data reuse information.

5.2.2.3 Step 3: computing thread affinity using an affinity graph

Once an affinity graph is constructed for an application and for a given number of threads, we can use it to investigate multiple thread pinning strategies. The idea is based on graph partitioning methods [KK98c]. The affinity graph must be decomposed into disjoint subsets,

$$\begin{cases} \mathcal{V} = \{T_1, T_2 \cdots, T_n\} \\ \max \sum_{T_i,T_j,\in\mathcal{V}} Assign(T_i, T_j) \times \alpha(T_i, T_j) \quad T_i \neq T_j \\ \text{Subject to} \\ \sum_{T_j \in \mathcal{V}} Assign(T_i, T_j) = 1, \quad \forall T_i \in \mathcal{V} \\ \sum_{T_i \in \mathcal{V}} Assign(T_i, T_j) = 1, \quad \forall T_j \in T \\ Assign(T_i, T_j) \in \{0, 1\} \quad \alpha(T_i, T_j) \in \mathbb{N} \quad \forall T_i \neq T_j \in \mathcal{V} \end{cases}$$

Figure 5.3: Formal mathematical definition of the linear assignment problem of threads

named a partition. A partition $V = \{V_1, V_2, \cdots, V_k\}$ has the property that $\bigcup_{1 \leq l \leq k} V_l = \mathcal{V}$ and $V_l \cap V_m = \emptyset$, where $l \neq m$ and $l, m \in [1, k]$. Every subset $V_l \in V$ contains a set of nodes representing threads that have to be placed on adjacent cores sharing the same cache level (L2 or L3, depending on the target machine). If we have k shared caches on the system, then we compute a partition with k subsets [KK98c]. The global objective function is to maximise $\sum_{(T_i,T_j) \in V_l \times V_l} \alpha(T_i, T_j)$ the sum of the gains between threads belonging to the same partition. Graph partitioning is a classical NP-complete problem, so we have to use a heuristics such as [KK98c]. Fortunately, we have a special polynomial case explained below.

LP technique: Partitioning the affinity graph into pairs of threads

If we are faced to a machine architecture where a cache level is shared between two adjacent cores (such as in the `Core2` machine), then the problem becomes to compute partitions with a size equal to 2 ($\|V\| = 2$). It is easy to see that in the case of partitions of size 2 the problem is equivalent to computing a set of thread pairs sharing a common cache while maximising a global gain. In this special case, the optimisation problem can be solved with a simpler maximum-weight matching in general graphs [Edm65]. Precisely, it can be polynomially and optimally solved thanks to the algorithm of Edmonds in $O(\|\mathcal{V}\|^2.\|\mathcal{E}\|)$ [Edm65].

Due to the lack of an efficient implementation of the algorithm of Edmonds [Edm65], we decided to use an existing implementation of a distinct algorithm that allows us to compute thread pairs while optimising a global gain function[1]. Algorithm 1 gives an abstract view of computing a set of thread pairs from a graph $G = (\mathcal{V}, \mathcal{E})$. Computing for thread pairs may be modelled by a linear assignment problem [Kuh55]. This simplification does not compute exactly the optimal solution of a maximum-weight matching in general graphs (algorithm of Edmonds [Edm65]) but simplifies our technical implementation. Therefore, the problem of computing thread pairs can be expressed as following: given n threads, solving the linear assignment problem consists in finding thread pairs that maximise the total data block share while ensuring that each thread is assigned to only one different thread. For each pair of threads T_i and T_j, the variable $Assign(T_i, T_j)$ is equal to 1 if the thread T_i is assigned to thread T_j, or to 0 otherwise. Solving this assignment problem produces thread pairs, every pair of threads is supposed to have a significant data reuse and sharing. If we execute every pair of threads on adjacent cores sharing a common cache level, we hope to enhance cache utilisation between the two threads (less cache misses, less use of memory bandwidth). Figure 5.3 formally defines the linear assignment problem computing the pairs of threads.

Our mathematical model computes a thread affinity using an objective function that maximises the inter-thread data sharing. However, this model does not define a cost model that can

[1]Instead of using an implementation of the maximum-weight matching algorithm, we decided to solve the problem by a Branch and Bound algorithm using the `lpsolve` framework

predict a possible performance degradation as a consequence of running the application with the computed thread affinity. Since thread affinity is computed upon a data sharing metric, we consider that if a computed thread affinity is inefficient (in terms of program execution times), then this means that: 1) data reuse is not effectively exploited (the reuse distance between threads is too long) by that thread affinity, 2) the amount of data sharing in the application is not important or 3) the computed thread affinity has to account for other factors than data sharing (bandwidth saturation, memory pages allocation, etc.). For this reason, to analyse the effectiveness of a computed thread affinity, we use the metrics presented in Section 5.2.3.

Algorithm 1 ComputePairs(Graph : G(V,E))

Require: G
  $s \leftarrow |E|/2$
  if $|E|$ mod $2 = 0$ **then**
    {Optimal thread pairs are computed by solving the modified version of the linear assignment problem of threads}
    return $C = \{C_0, C_1, ..., C_s\}$ {return the set of the thread partitions}
  end if

Figure 5.4 shows an example of a parallel application using four threads, its α matrix is given by Table 5.4a. Solving the problem for this α matrix produces the assignment matrix given by Table 5.4b. A value of an element in the assignment matrix $Assign(i, j)$ equal to 1 means that a matching is found between the thread T_i in row i and the thread T_j in column j. A zero value in the assignment matrix means that no matching was found. In this example the algorithm produces the pairs (T_1,T_2) and (T_3,T_4). These two thread pairs are supposed to define threads with strong data reuse relationship. We hope that executing every thread pair on two adjacent cores sharing the same cache level would improve global application performance (since cache effects between threads would be improved).

	T_1	T_2	T_3	T_4
T_1	0	923177138	909675518	916697725
T_2	923177138	0	926145460	914237540
T_3	909675518	926145460	0	940029712
T_4	916697725	914237540	940029712	0

(a) $\alpha(T_i, T_j)$

	T_1	T_2	T_3	T_4
T_1	0	1	0	0
T_2	1	0	0	0
T_3	0	0	0	1
T_4	0	0	1	0

(b) Assignment matrix

Figure 5.4: Thread pair computation by solving the thread assignment problem

The current section explained how to optimally compute thread pairs to be executed on two adjacent cores sharing the same cache level. We call this thread pinning technique as `LP`. In many architectures, some cache levels are shared between more than two cores. This requires to partition the affinity graph as explained in the beginning of Section5.2.2.3. Since this problem is NP-complete, the next section explains our different heuristics.

GP technique: Partitioning the affinity graph into groups of more than two threads

In Section 5.2.2.3, we proposed an approach (called `LP`) based on solving the problem of linear assignment of threads to produce optimised pairs of threads. This approach is convenient in the case of a multicore processor cache architecture where each couple of cores share a single cache level. If more than two cores share a cache, we present here another approach based on

graph partitioning.

If we consider an undirected graph $G = (\mathcal{V}, \mathcal{E})$, the objective of the traditional graph partitioning problem is to compute a balanced k-way ($k > 1$) partitioning such that the number of edges (or in the case of a weighted graph, the sum of their weights) that straddle different partitions is minimised. Assuming that P is a vector of size $\|\mathcal{V}\|$ such that $P[i]$ stores the number of the partition that vertex i belongs to. The *edgecut* of this partitioning is defined as the number of edges that straddle partitions. That is, the number of edges (v, u) for which $P[v] = P[u]$. If the graph has weights associated with the edges, then the edgecut is defined as the sum of the weights of these straddling edges [KK98a]. In our study, we consider the affinity graph $G = (\mathcal{V}, \mathcal{E}, \alpha)$ defined in Section 5.2.2.2 where $\mathcal{V}$ is the set of the application threads, $\mathcal{E}$ is the set of edges between each pair of threads and where the weight α represents the data block sharing between threads. Algorithm 2 shows an abstract view of partitioning a graph $G = (\mathcal{V}, \mathcal{E})$ into multiple partitions.

Algorithm 2 Partition(Graph : G(V,E), number of partitions : r)

Require: G, r
  {We do not need to partition the graph if the number of vertices is equal to the number of partitions}
  if $|G| > r$ **then**
    {The partitions are computed using the multilevel recursive bisection algorithm}
    return $P = \{P_0, P_1, ..., P_r\}$ {return the set of the thread partitions}
  end if

Each partition of the affinity graph represents the set of threads to be placed on cores sharing the *same* cache level. Consequently, the number k of partitions to compute is equal to the number of available shared caches at a certain level(L2, L3, `NUMA` nodes, etc.). In the current section, we assume that the number of cores that share a common cache must be greater than 2. The previous section studied the special case where only two cores share a common cache.

We use the `METIS` [KK98b] software package to achieve graph partitioning. `METIS` offers a set of routines allowing to partition graphs and meshes using different algorithms. The partitioning algorithms implemented in `METIS` are based on multilevel graph partitioning described in [KK98a, KK98c, KK98b].

Among the various routines provided by `METIS`, we are interested mainly by two partitioning routines: `METIS_PartGraphRecursive` and `METIS_PartGraphKway`. `METIS_PartGraphRecursive` is used to partition a graph into k equal-size parts using multilevel recursive bisection described in [KK98a]. `METIS_PartGraphKway` is used to partition a graph into k equal-size parts using the multilevel k-way partitioning algorithm described in [KK98c]. Both of these routines are able to produce high quality partitions. However, depending on the application, one program may be preferable than the other. In general, `METIS_PartGraphKway` is preferred when it is necessary to partition graphs into more than eight partitions. On the other hand, `METIS_PartGraphRecursive` is preferable for partitioning a graph into a small number of partitions as advised by the `METIS` development team.

The previous routines require to specify as an argument the number of graph partitions to compute. When the partitioning finishes, each vertex in the graph is assigned to one of

the computed partitions. Figure 5.5 shows an example on how the `METIS` routines are used to partition a graph of eight threads. In this example, the partitioning algorithm is invoked to produce two parts from the original graph.

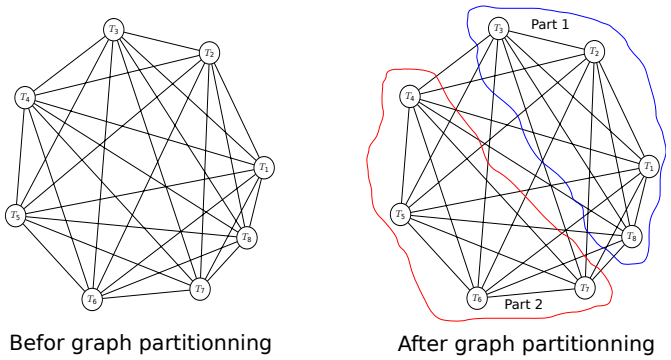


Figure 5.5: Partitioning an affinity graph of eight threads into two parts. Each part corresponds to four threads that must be executed on adjacent cores sharing the same cache.

The two previous sections presented two methods for computing a good thread pinning. They both optimise data locality and sharing between threads at a certain cache level only (say L2 or L3). The next section presents another approach that tries to compute a thread pinning which optimises data locality and sharing at multiple cache levels.

LPGP and GPLP techniques: Combining the LP and the GP techniques

This section presents a hybrid strategy to compute optimised thread pinnings. This approach combines the LP method with the GP one. We developed this method to compute thread affinities that optimise data locality between threads at multiple cache levels. Depending on the method by which we start (either LP or GP) the computation of an affinity, we can consider the following strategies: LPGP and GPLP.

Let us start by explaining the LPGP technique. It is summarised by the following steps:

1. We start by a step of optimal computation of a set of thread pairs from an affinity graph $G = (\mathcal{V}, \mathcal{E}, \alpha)$ using the LP method as explained in section 5.2.2.3. Two threads inside a pair must be placed on two adjacent cores. This first step aims to optimise data sharing at a fine-grain (pair) granularity.

2. After we have computed the set of thread pairs, we compute a new affinity graph $G' = (\mathcal{V}', \mathcal{E}', \alpha')$. In G', $\mathcal{V}'$ represents the set of thread pairs, and $\mathcal{E}'$ represents the relationship between each pair of threads. The affinity graph G' has a gain function (weight) α' associated with each edge. An edge $e' = (P_1, P_2) \in \mathcal{E}'$ represents the affinity between two thread pairs $P_1 = (u_1, v_1) \in \mathcal{V}'$ and $P_2 = (u_2, v_2) \in \mathcal{V}'$. We compute $\alpha'(e')$ as follows:

$$\alpha'(e') = \alpha(u_1, u_2) + \alpha(u_1, v_2) + \alpha(v_1, v_2) + \alpha(v_2, v_2)$$

3. Now, we have the new affinity graph G', we can proceed by a hierarchical k-partitioning as as presented in section 5.2.2.3. The number k' of partitions at each partitioning level is equal to the number of shared caches at that level. For instance, if we have a machine with 2 NUMA compute nodes, each having 4 processors. If we consider also that each

processor has 4 cores sharing a common L3 cache, and where each couple of cores share a common L2, then we can consider the following hierarchical k-partitioning. First, G' is partitioned into 2 partitions (NUMA nodes level), this step fixes thread pairs into each NUMA node. Second, each partition from G' is in turn partitioned into 4 sub-partitions (processor/L3 level). The later step fixes thread pairs on each processor. Finally, each thread pair inside a processor is fixed on cores according to L2 sharing topology between cores.

We defined another hybrid method called `GPLP`. This strategy starts by a hierarchical k-partitioning of the affinity graph until the last level in the memory hierarchy is reached (shared caches between pair of cores). When the last level is reached and for each computed partition, we use the polynomial method to compute optimised pairs of threads for that partition in such a way we can assign the pairs to neighbours cores with shared caches. It is clear that this strategy is only effective in the case of target machines where memory hierarchy has at least two levels and where the last level must be shared caches between each couple of cores.

After presenting the `LPGP` and the `GPLP` strategies above, we can give the algorithms of these two techniques. Before, let see below the definition of terms we need :

1. $Core_i$ is the i^{th} core of the machine.
2. L_i^j is the j^{th} cache of level i. The first level is 1. The root level is 0. The leafs represent compute cores.
3. `Share(`L_i^j`)` is the set of cores sharing the cache level L_i^j.
4. `Height(T)`: returns the length of the longest downward path to a leaf from the root in the tree T.
5. `Width(T,l)` returns the number of nodes in the level l of the tree T.
6. `Child(`L_i^j`)` returns the number of child nodes of the node L_i^j.
7. `Graph(E)` builds a complete graph from the set E.
8. `Threads(P)` builds a set of threads in the partition P.

Now, that we have all the definitions, Algorithms 4 and 5 give the pseudo-code of the `LPGP` and `GPLP` strategies. Besides, we have Algorithm 3 which shows in pseudo-code how threads are binded to the machine cores. We have to notice that we represent a machine as a tree data structure where each level in the tree corresponds to a different level in the memory hierarchy (Node, socket and cache level).

Summary of the application dependent thread pinning techniques

Based on the approaches presented above, we can define the following thread affinity strategies, corresponding to the application of heuristics for solving graph k-partitioning problems at each level of the memory cache hierarchy of the parallel machine:

1. `GP` strategy. Apply a graph k-partitioning only to place threads on sockets. For instance, on the `Nehalem` machine, we compute two partitions since we have two shared L3 caches (one L3 per socket).

Algorithm 3 Assign(Partition of threads: Tp, Set of cores : C)

Require: Tp, C
  if $|Tp| > |C|$ **then**
    {The number of cores must be less or equal to the number of threads}
    $STOP$
  end if
  for $i = 1 \rightarrow |Tp|$ **do**
    {Set the affinity of thread Tp_i to core C_i}
    $set\_affinity(Tp_i, C_i)$
  end for

Algorithm 4 LPGP(Affinity Graph : G(V,E), The machine tree : M)

Require: G, M
  $G' \leftarrow ComputePairs(G)$
  $H \leftarrow Height(M)$
  {Go through all the levels of the machine tree M}
  for $i = 1 \rightarrow H - 1$ **do**
    if $i = H - 1$ **then**
      {Stop the algorithm when we reach the L2 caches level}
      for $p = 1 \rightarrow Width(i)$ **do**
        {Assign the p^{th} pair of threads to the pair of cores in L_i^p}
        $Assign(Threads(P_i^p), Share(L_i^p))$
      end for
      return
    end if
    if $i = 1$ **then**
      {Perform the first partitioning of the affinity graph G'}
      {The number of partitions is equal to the number of nodes at the first level of the memory hierarchy}
      $Partition(G', Child(root))$
      for $k = 1 \rightarrow Child(root)$ **do**
        {Assign the k^{th} partition of threads to the set of cores in L_i^k}
        $Assign(Threads(P_i^k), Share(L_i^k))$
      end for
    else
      for $j = 1 \rightarrow Width(i-1)$ **do**
        {Partition the affinity graph for the j^{th} cache of level $i-1$}
        $Partition(Graph(P_{i-1}^j), Child(L_{i-1}^j))$
      end for
      {Cross all the nodes at level i and}
      {assign the k^{th} partition of threads to the set of cores in L_i^k}
      for $k = 1 \rightarrow Width(i)$ **do**
        $Assign(Threads(P_i^k), Share(L_i^k))$
      end for
    end if
  end for

Algorithm 5 GPLP(Affinity Graph : G(V,E), The machine Tree : M)

Require: G, M
  $H \leftarrow Height(M)$
  for $i = 1 \rightarrow H - 1$ **do**
    if $i = 1$ **then**
      {Perform the first partitioning of the affinity graph G}
      {The number of partitions is equal to the number of nodes at the first level of the memory hierarchy}
      $Partition(G, Child(root))$
      for $k = 1 \rightarrow Child(root)$ **do**
        {Assign the k^{th} partition of threads to the set of cores in L_i^k}
        $Assign(Threads(P_i^k), Share(L_i^k))$
      end for
    else
      for $j = 1 \rightarrow Width(i-1)$ **do**
        if $i = H - 1$ **and** $Child(L_i^j) = 2$ **then**
          {Compute thread pairs if the cache is shared by pair of cores}
          $ComputePairs(Graph(P_{i-1}^j))$
        else
          {Partition the affinity graph for the j^{th} cache of level $i-1$}
          $Partition(Graph(P_{i-1}^j), Child(L_{i-1}^j))$
        end if
      end for
      for $k = 1 \rightarrow Width(i)$ **do**
        {Cross all the nodes at level i and}
        {assign the k^{th} partition/pair of threads to the set of cores in L_i^k}
        $Assign(Threads(P_i^k), Share(L_i^k))$
      end for
    end if
  end for

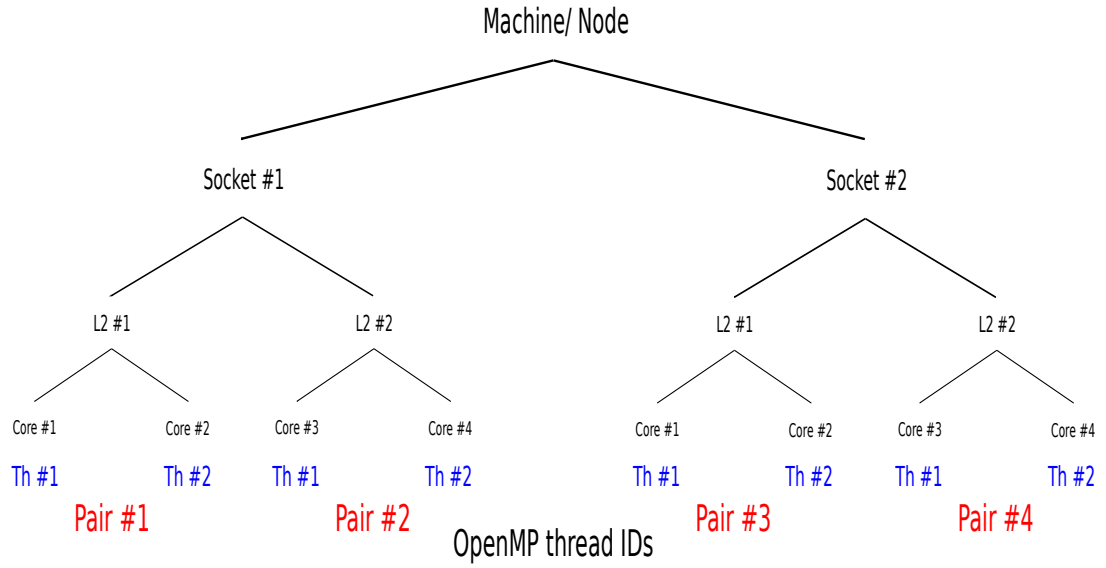


Figure 5.6: Thread placement following the `LP compact` strategy on a machine with two sockets, each socket has four cores where each pair of cores shares an L2 cache

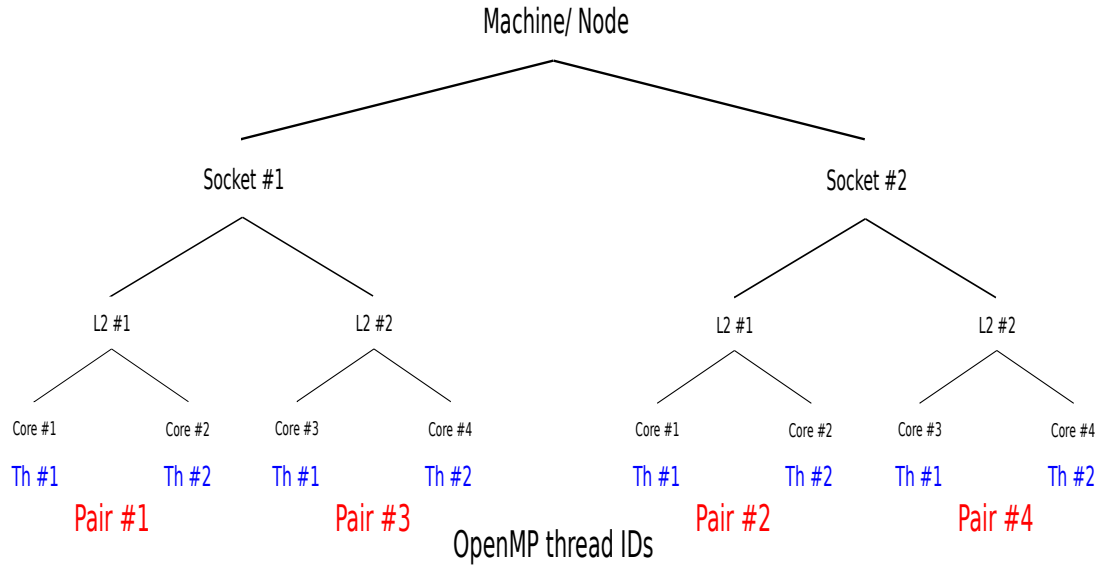


Figure 5.7: Thread placement following the `LP scatter` strategy on a machine with two sockets, each socket has four cores where each pair of cores shares an L2 cache

2. `LP compact` strategy (Figure 5.6). After using the polynomial method to optimally compute a set of thread pairs, this strategy assigns successive thread pairs (in order of their computation) to cores with shared caches as close as possible.

3. `LP scatter` strategy (Figure 5.7). After using the polynomial method to optimally compute a set of thread pairs, it distributes the thread pairs as evenly as possible across the entire set of sockets of the machines (one thread pair per socket if possible).

4. `LPGP` strategy (Figure 5.8). After an initial step of optimal computation of thread pairs,

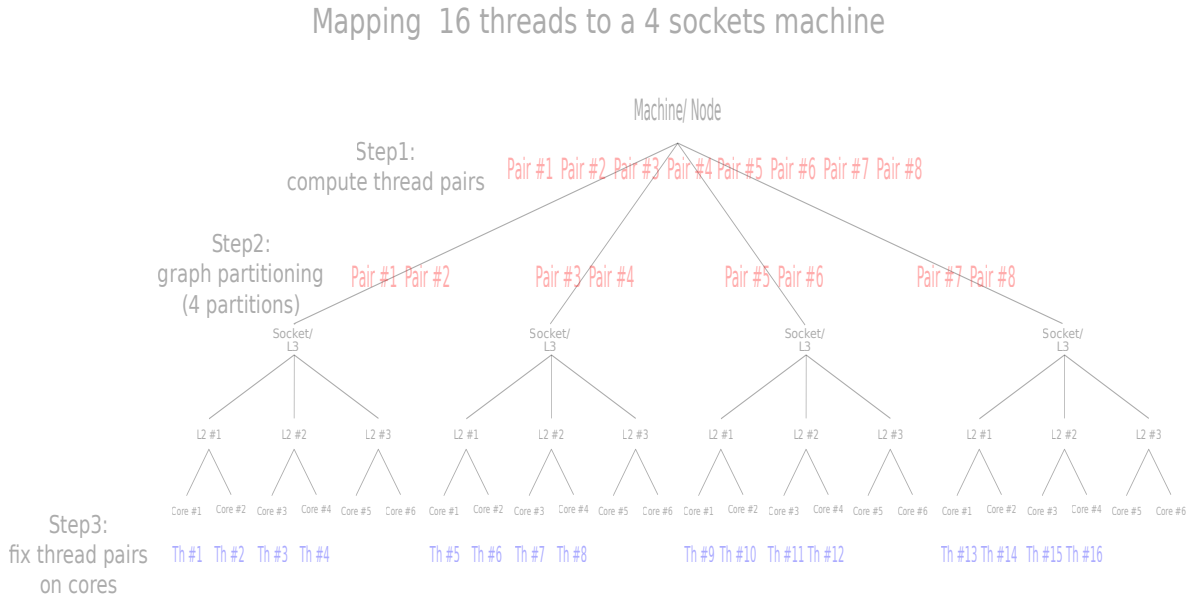


Figure 5.8: Thread placement of an application running with 16 threads following the `LPGP` strategy on a machine with four sockets, each socket has six cores sharing an L3 cache and where each pair of cores shares an L2 cache

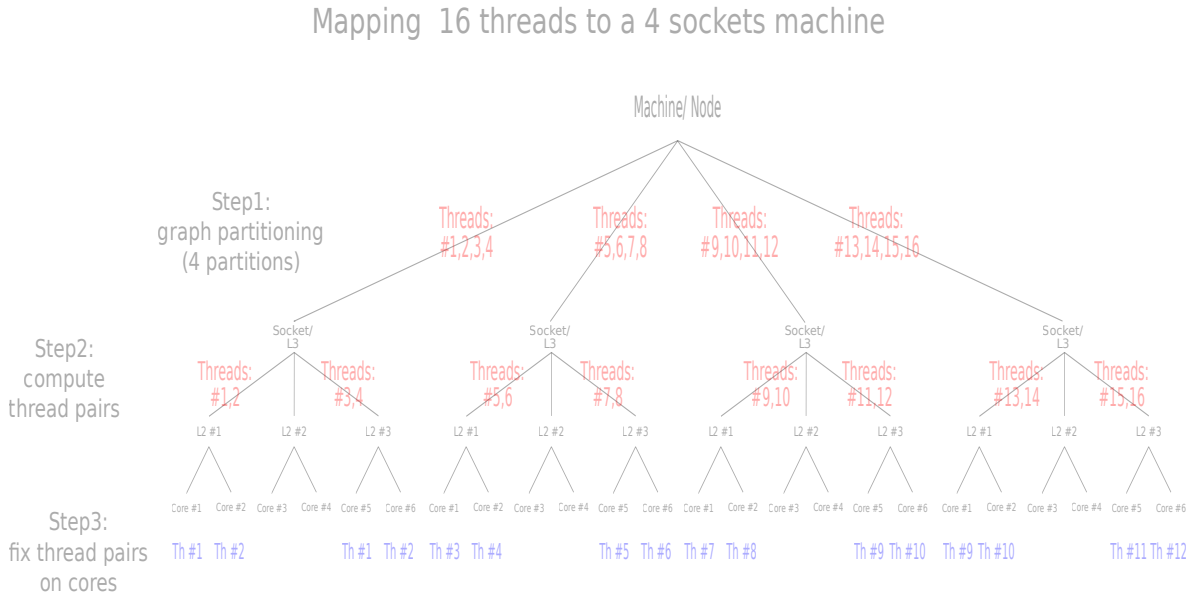


Figure 5.9: Thread placement of an application running with 16 threads following the `GPLP` strategy on a machine with four sockets, each socket has six cores sharing an L3 cache and where each pair of cores shares an L2 cache

we proceed by a graph k-partitioning [KK98c]. It is a hierarchical strategy, where threads are first paired and pinned on shared L2 or L3 cache then thread pairs are partitioned

and placed on the different sockets according to their affinity.

5. `GPLP` strategy (Figure 5.9). It is a hierarchical strategy. It starts by an initial graph k-partitioning to fix threads on sockets, then perform an optimal polynomial algorithm to compute thread pairs sharing L2 or L3 cache levels.

The next section presents some metrics that we use for inter-thread data sharing characterisation.

5.2.3 Metrics for data sharing characterisation

Most often, it is necessary to quantify the amount of sharing that a given application implements. For this purpose, we define two metrics which are deduced from the memory trace analysis: 1) the working set size and 2) the data reuse ratio. These metrics are detailed bellow.

5.2.3.1 The working set size

Although not accurate, comparing the amount of data accessed by each thread to the total amount of data accessed by the whole application, may give an indication about the degree of data sharing in the application. To do so, we can distinguish between:

- The total working set of the application: it is equal to the size of all unique 64 bytes data blocks accessed by all the threads
- The working set of each thread: is equal to the size of all unique 64 bytes data blocks accessed by each thread. We have to notice that in this per thread view of the working set, the 64 bytes data blocks may be counted multiple times (once for each thread) if they are accessed by multiple threads.

5.2.3.2 The data reuse ratio (DRR)

Most often it is necessary to quantify the amount of sharing that a given application implements. For this purpose, we define the data reuse (share) ratio (DRR). If we consider T the set of threads, P the set of pairs computed using the maximal-weight matching graph algorithm where $|P| = \frac{|n|}{2}$, $Lines(p)$ the number of touched data blocks by the pair p and $Access(p)$ the number of accesses to the data blocks by the pair p then, the reuse ratio can be calculated by the following formula :

$$DRR = \frac{\sum Lines(P_i) \times Access(P_i)}{\sum Lines(T_j) \times Access(T_j)} \times 100 \quad \forall P_i \in P \quad \forall T_j \in T$$

To quantify the inter-thread sharing of parallel programs using the DRR metric, we define three interval values highlighting the sharing degree. We mainly observed these values experimentally[2]:

1. No sharing : $DRR < 1\%$.
2. medium sharing : $1\% \leq DRR < 4\%$
3. good sharing : $DRR \geq 4\%$

The next section presents the experimental setup that we use and the results of our performance evaluation and analysis.

[2]These values are somehow dependent on our context. These three intervals can be different in other experimental contexts

5.3 Experimental setup and methodology

This section describes the experimental setup that has been used to conduct the study. We describe the benchmarks, the hardware platforms and the experimental methodology.

5.3.1 Software environment

Our experiments have been conducted using all SPEC OMP01 [Sta06] NAS Parallel Benchmarks (NPB) [JFY99]. For all the OpenMP applications, we used both the `train` and `ref` data inputs in SPEC and the `Class B` in NPB. We tested multiple number of threads for every application according to the number of available of cores. We tested nine thread placement strategies for every application, thread number, input data set, while repeating the execution 35 times (for statistical significance analysis).

The benchmarks have been compiled using three different compilers (gcc 4.1.3, gcc 4-3.2, icc 11.1) with flag `-O3 -openmp`. In this thesis, we report the performance numbers obtained using the vendor compiler icc 11.1 because it provided the best performance in overall. All the tested machines run the Linux kernel.

5.3.2 Hardware setup

We conducted all our experiments on three platforms:

1. The `Core2` (8 cores) machine (Figure 5.10). It is a single SMP machine with two `Clovertown` sockets (Intel Xeon E5345 with the `Core2` micro-architecture). Each processor has 4 cores, each pair of cores have a shared L2 cache. The platform has two L2 4MB caches per socket, for both instructions and data. The core frequency is 2.33 GHz. Each core has a separate 32KB L1 data and instruction caches. The main memory size is 4 GB RAM. The front-side bus has a clock rate of 1.33 GHz.

2. The `Nehalem` (8 cores) machine (Figure 5.11). It is a single SMP machine with two `Gainestown` sockets (Intel Xeon X5570 with the `Nehalem` micro-architecture). Each processor has 4 cores with a shared L3 cache. The platform has two L3 caches of 8 MB, one on each chip, for both instructions and data. The core frequency is 2.93 GHz. Each core has a private 256KB L2 cache unified for data and instructions. In addition, each core has a separate L1 data and instruction caches with 32 KB each. The main memory size is 24 GB. Each chip in the platform has an integrated memory controller.

3. The `ccNUMA` (96 cores) machine (Figure 5.12). It is an IBM System X3950M2 `ccNUMA` shared memory machine with four compute nodes. Each node has four `Dunnington` sockets (Intel Xeon X7460 with the `Core2` micro-architecture). Each socket (chip) has 6 cores, where each pair of cores has a shared 3MB L2 cache. The 6 cores of a chip have a shared 16MB L3 cache. The L2 and L3 caches are unified for data and instructions. Each core has a separate L1 data and instruction caches of size 32 KB. The core frequency is 2.66 GHz. Each node has a quad 1066 MHz FSBs (one per socket) and a 47 GB RAM memory domain (188 GB in total). In this platform, 256 MB of virtual cache per node is used for interprocessor communications between nodes, to keep data in synchronisation (this amounts to as much as 1 GB in total taken from main memory).

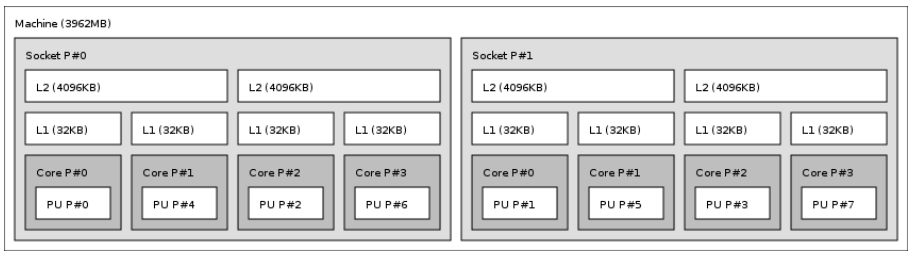


Figure 5.10: `Core2` SMP machine architecture

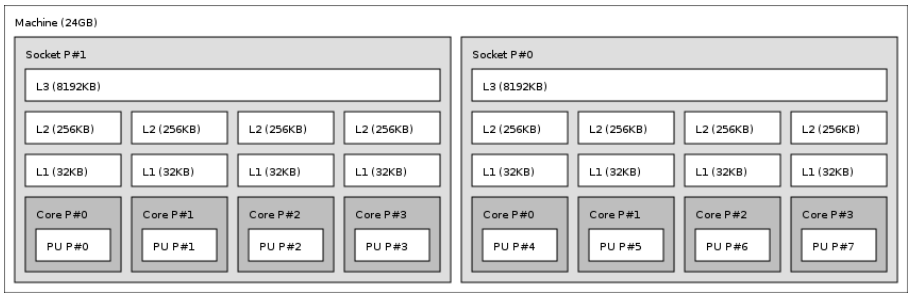


Figure 5.11: `Nehalem` SMP machine architecture

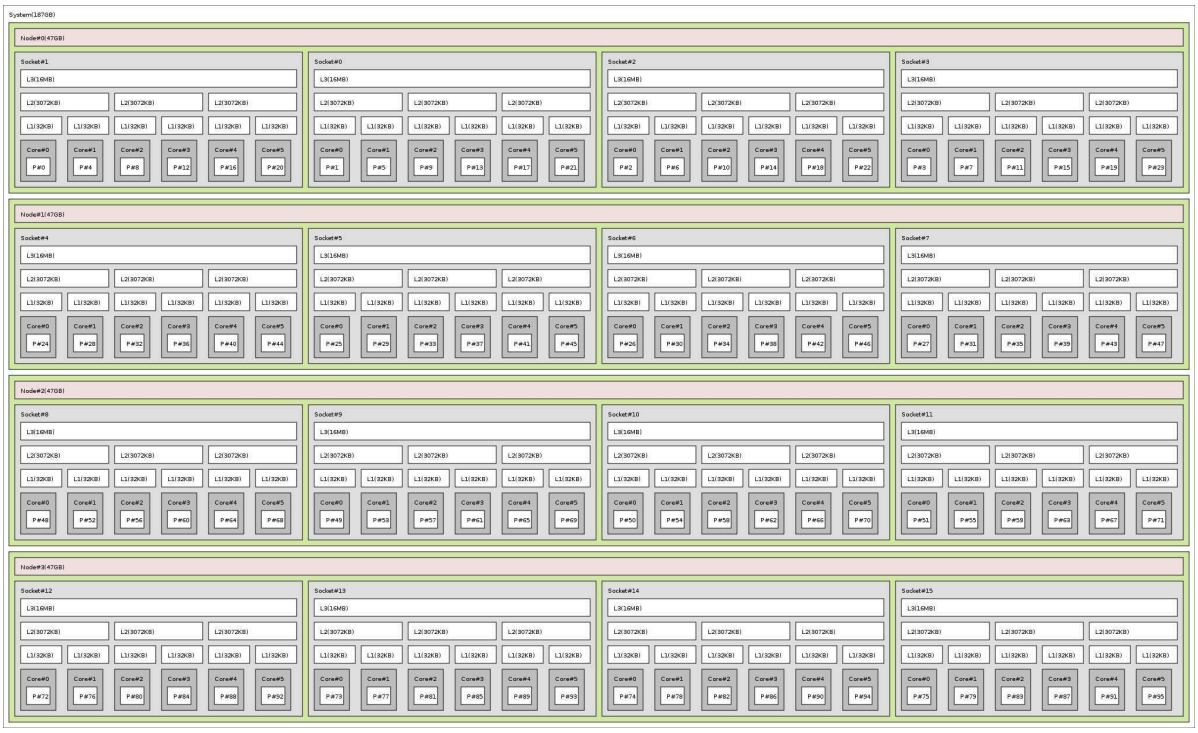


Figure 5.12: `ccNUMA Core2` machine architecture

5.3.3 Experimental methodology

In order to improve the reproducibility of the results, the experiments were done following some practices:

- The test machines were entirely dedicated during the experiments to a single user when it is possible.

- Running each benchmarks 35 times [Raj91, TWB10] for each software configuration. This high number of runs allows us to report statistics with a high confidence level;
- The repetitive executions of the same application were performed sequentially in a back-to-back way;
- We use small memory pages (4 KB).
- All hardware prefetchers are enabled.
- The experiments were done on a minimally-loaded machine if possible. This is true only on the `Core2` SMP and in the `Nehalem` machines;
- All inessential shell environment variables are unset;
- Deactivation of the randomisation of the starting address of the stack (this is an option in the Linux kernel versions since 2.6.12). This is achieved only on the `Core2` SMP machine;
- Dynamic voltage scaling disabled;
- The Intel `Turbo Boost` and `Hyper-Threading` technologies were disabled on the `Nehalem` SMP;
- Using the build system and scripts of OMP2001 to compile and optimise the applications, launch them, measure execution times, check validity of the results and report the performance numbers for the experiments;
- All the measurements of the execution times for the three machines rely on the `gettimeofday` function;
- No more than one application was executed at a time.
- We use violin plots to report the observed 35 execution times of each application in each software configuration. Violin plots are similar to box plots, except that they also show the probability density of the data at different values. The white dot in each violin gives the `median` and the thick line through the white dot gives the inter-quartile range.

5.3.4 Statistical significance analysis

When faced to variations in the observed execution times, we must use rigorous statistics to study the validity of our empirical conclusions. Empirical conclusions must not rely on sample metrics such as sample means or medians [Raj91], we must rely on statistical tests. This is done thanks to the Speedup-Test methodology described in [TWB10] (see Section 3.1.3.2 for a quick recall). Declaring a statistical significance of a speedup (either for mean or median execution times) follows a formal protocol:

- Comparing between the average execution times of two samples (two thread placement strategies) is done thanks to the one-sided Student t-test.

- Comparing between the median execution times of two samples (two thread placement strategies) is done thanks to the one-sided Wilcoxon-Mann-Whitney test.

Let us define $X = x_1, \cdots, x_n$ and $Y = y_1, \cdots, y_m$ as two finite sets of measurements of program program execution times, where X represents a baseline configuration and Y an optimised configuration. n and m are the sizes of X and Y respectively with n possibly different from m. x_i and y_i represent the measurement of program execution time of the i^{ith} execution in X and Y respectively. Having all these constraints, we can compute two types of speedups :

1. The observed speedup of the mean (average) execution times :

$$spmean = \frac{mean(X)}{mean(Y)}$$

2. The observed speedup of the median execution times :

$$spmedian = \frac{med(X)}{med(Y)}$$

The next section presents and analyses the performance results of the tested thread pinning techniques on the three experimental machine platforms.

5.4 Experimental study of the performance of the tested thread pinning techniques

This section introduces the performance evaluation of the application independent and application dependent thread affinity strategies. The analysis considers two aspects: 1) the performance enhancement and 2) the performance stability. We conduct our performance evaluation using a variety of parallel OpenMP applications. The used benchmarks are from SPEC OMP2001 and from Nas Parallel Benchmarks (implemented with OpenMP API). We also used three different machines showing three typical multicore architectures in the market, two machines are single SMP machines with 8 cores (`Nehalem` and `Core2` micro-architectures) and a 96 cores (`Core2` micro-architecture) `ccNUMA` machine with a global shared memory. The detailed presentation of the benchmarks and the architecture machines is presented in section 5.3. Moreover, we varied the number of threads generated at runtime to be 4, 6 and 8 on the single SMP machines and we experimented the 16 and 96 threads cases in the `ccNUMA` machine (96 cores).

Benchmark	4	8	96
wupwise	600X	420X	490X
swim	60X	35X	25X
mgrid	530X	330X	100X
applu	300X	250X	130X
galgel	890X	350X	90X
equake	670X	260X	130X
apsi	350X	180X	140X
fma3d	240X	190X	80X
art	1000X	1000X	170X
ammp	230X	450X	180X

Table 5.1: Performance overhead of memory trace analysis in SPEC OMP01 when 4, 8 and 96 threads are used.

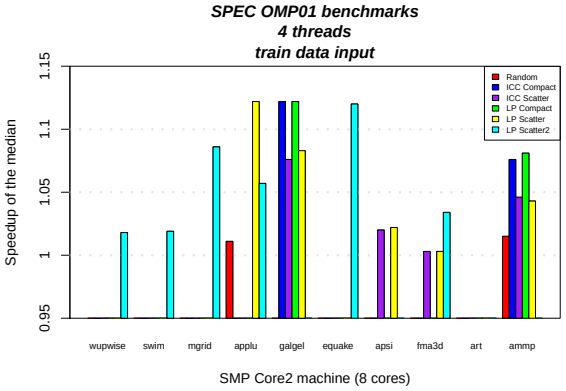
Before presenting the detailed performance analysis, let us show the overhead inherent to memory accesses tracing. Table 5.1 reports the slowdown factors when the number of threads is 4, 8 and 96 compared to a native execution. It indicates that the cost of the profile execution is significant with slowdown factors are in the range of 25X and 1000X.

5.4.1 SPEC OMP2001 benchmarks

We started our performance evaluation with SPEC OMP2001 benchmarks to evaluate the impact of thread affinity on the overall performance of parallel multi-threaded applications. We run each application from SPEC OMP2001 with 4, 6 and 8 threads on the SMP machines using

the `train` data input and with 16 and 96 threads in the ccNUMA machine using the `ref` data input.

5.4.1.1 SMP machines results (`Core2` and `Nehalem`)



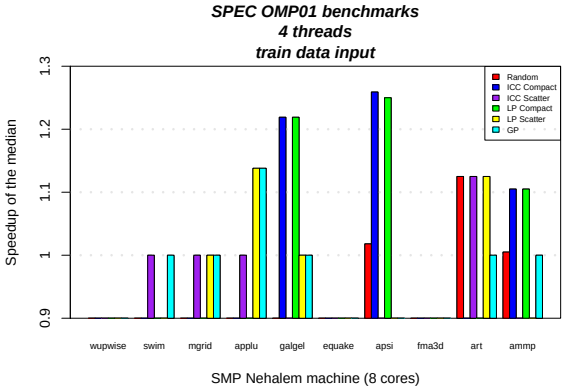
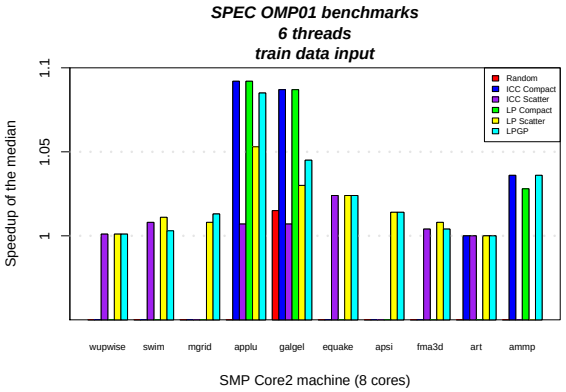
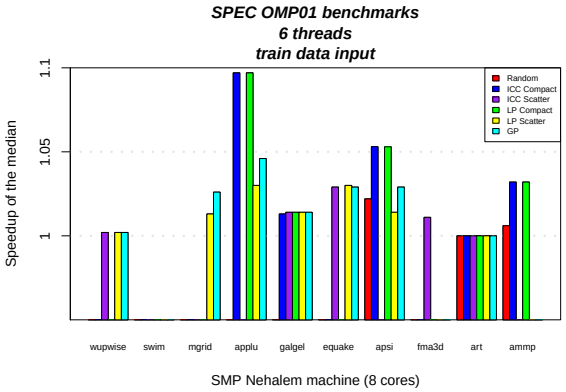
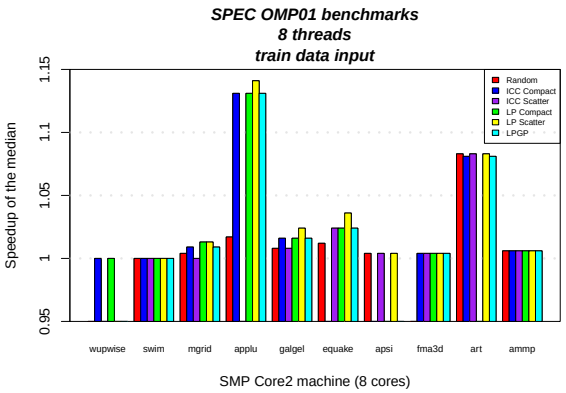


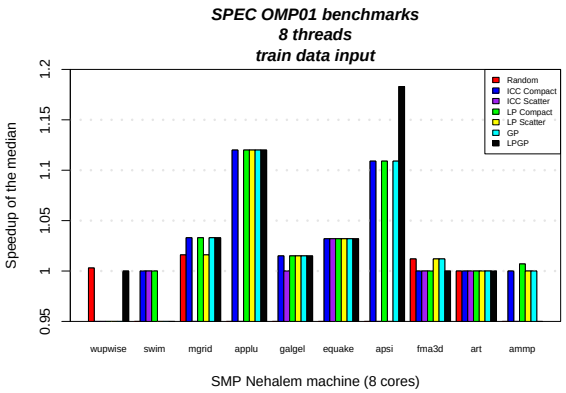
Figure 5.13: The observed speedups of the median execution times on the `Core2` SMP machine

Figure 5.14: The observed speedups of the median execution times on the `Nehalem` SMP machine









Running SPEC OMP01 applications without affinity on the `Core2` machine produces a relative variability (RV) in the range $2-27\%$ with 4 threads, $4-23\%$ with 6 threads and in the range $0-4\%$ with 8 threads. Setting thread affinity (either application dependent or application independent affinity) produces in general stable performance. We observed a relative variability in the range of $\approx 0-2\%$ in almost all the benchmarks. On the `Nehalem` machine, the observed

RV is negligible in almost all the tested thread affinity strategies (less than 4%). However, these observations are not true for the `Random` strategy in both machines, this configuration exhibits poor performance stability, it is in the range range $2-26\%$ on the `Nehalem` for instance. Indeed, although this strategy prevents thread migration, the initial random placement of threads is important and wrong pinning leads to significant variations of program execution times. This situation is predictable since this strategy experiments many cache sharing configurations.

Figures 5.13 and 5.14 report the observed sample speedups of the median execution times for both machines. We have to notice that we plot only statistically significant speedups[3]. Each speedup in these figures represents the median execution time of the tested pinning technique normalised to the median execution times of the `no affinity` configuration. So, a bar higher than 1 means that the tested pinning technique is better than the OS free strategy. As we can see, the program performance behaviour of SPEC OMP applications running on the `Nehalem` and the `Core2` machines are quite similar. Indeed, the analysis of the observed speedups and the amount of data sharing implemented in each application allowed us to consider three classes of benchmarks exhibiting: small data share, medium data share and significant data share.

When the number of threads does not exceed the number of cores, the contention caused by sharing the last level cache makes that it is better to run the `mgrid`, `wupwise`, `swim`, `fma3d`, `equake` and to some extent the `art` benchmark in separate sockets if possible to achieve decent and stable performance. Moreover, mainly these benchmarks exhibit less inter-thread data sharing. On the other hand, sharing is beneficial for benchmarks like `galgel`, `ammp` and to some extent the `apsi` benchmark. This performance behaviour is also confirmed by our data sharing metrics regarding the amount of inter-thread data sharing implemented on these benchmarks. Finally, we can see that mostly, the `applu` benchmark achieves better performance under the `LP scatter` strategy (35% reduction in the last level cache misses in the `Nehalem` machine for instance). This performance behaviour is explained as the following: 1) the later benchmark exhibits a medium data sharing, and 2) we know that the `LP scatter` strategy represents an intermediate situation between using full cache sharing and using separate sockets.

Let us now consider the case when the number of threads is equal to the number of cores. Although in terms of speedups, the application dependent strategies achieve statistically better speedups than application independent, we do not really observe any important difference. Except for some benchmarks where wrong pinning can lead to significant slowdowns, the observed performance behaviour can be explained by two main reasons: 1) the uniform distribution of working sets in SPEC OMP01 (it means small amount of data sharing with 8 threads), and 2) the working sets that in most cases exceed the size of shared caches, makes that the intensity of accesses and contention on shared caches is balanced. Therefore, we obtain similar program performance regardless of the tested thread pinning technique.

In this section, we shown the relation between the performance of SPEC OMP01 applications and thread affinity on SMP machines. The next section, shows if these applications behave similarly on a `ccNUMA` machine.

5.4.1.2 ccNUMA machine results

The `ccNUMA` machine is a 16 processors (96 cores) where each processor has 6 cores. The machine has 4 memory domains (nodes), each containing 4 processors. The exact topology of the

[3]We use the Speedup-Test protocol in order to compute statistically significant speedups.

machine is presented in Section 5.3. We run the SPEC OMP2001 benchmarks with 16 and 96 threads with different thread pinning strategies: `no affinity`, `Random`, `compact` and `scatter` of the `Intel` OpenMP runtime, `LP compact` and `LP scatter` and the finally the `LPGP` and `GPLP` hybrid strategies. All the applications were launched with the `ref` data input[4] (memory trace and performance evaluation).

We started our performance evaluation with the 16-threaded executions of the SPEC OMP01 suit. For each benchmark, we tested the eight thread affinity strategies cited above. We had an exception for the `LPGP` strategy where we computed three variants. The difference between them lies on the number of selected compute nodes in the target machine (of course, the number of used cores is still constant). Indeed, we computed eight optimised pairs using the polynomial method. After that, we applied the graph partitioning algorithm to compute four or two partitions where each partition has two or four pairs of threads(four or eight threads). Now, these partitions are pinned differently across the entire set of sockets of the machine:

1. Four partitions are pinned to four processors (a processor granularity) which are in the same node. In this pinning, all the threads in a given partition (4 threads) are sharing a common L3 cache.

2. Two partitions are pinned to 8 processors which are in two neighbours nodes (a node granularity). In this pinning, each pair of threads is pinned to a unique processor and share a common L3 cache.

3. Four parts are pinned to the 16 available processors (node granularity). This means that two pairs of threads inside a given part are in a distinct node. This time, a pair of threads are not pinned to one processor as usual but, to two neighbours processors which makes each thread runs in a distinct processor with an exclusive access to the large L3 cache.

Table 5.2 shows the observed relative performance variability when running with 16 threads. We can see that running the parallel applications without any thread affinity in a `NUMA` machine, leads to a significant variability ($\approx 60\%$ in the case of `swim`). The `Random` affinity is also interesting. Even if the observed RV is less than the one observed in the `no affinity` configuration, it is still important. This situation is predictable since `Random` exhibits various cache sharing configurations (without migration) in each run. Consequently, it leads to different observed program performance. Finally, we can see that all the remaining thread affinity strategies exhibit a small variability except in `apsi` and `equake`.

To compare the performance of the tested application independent and dependent thread pinning strategies against the `no affinity` strategy (base configuration), we report in Figure 5.15 the statistically observed speedups of the median execution times (using the Speedup-Test protocol). We observe in 6 out of 10 benchmarks that the program performance is better when each thread is pinned to one of the 16 distinct processors. This configuration ensures for each thread an exclusive access to the large L3 cache, reducing cache access contention and better single thread locality, hence, the observed good performance. This was observed in `wupwise`, `swim`, `mgrid`, `applu`, `equake` and `art`. The remaining benchmarks (`galgel`, `apsi`, `fma3d` and `ammp`) behave better when all the threads are pinned to a single node. This indicates that there is some data reuse and sharing between the threads which makes them taking benefit from sharing the common large L3. Furthermore, since the memory allocation follows the *first touch* policy, all the needed memory is allocated in the node where the threads are pinned. This

[4]We conducted experiments where we use the `train` trace analysis and `ref` for performance measurement. The obtained results remain similar.

Benchmarks	No affinity(%)	Random(%)	Other(%)
wupwise	41.90326	5.165399	< 1
swim	59.06282	46.33085	< 1
mgrid	27.54077	16.77693	≤ 1
applu	30.15992	11.68915	$0 - 6$
galgel	16.55328	12.41765	≤ 2
equake	23.16216	15.60925	$1 - 7$
apsi	37.37640	33.25014	$3 - 6$
fma3d	15.40391	14.3405	< 2
art	5.179218	3.865567	< 2
ammp	15.21677	14.36428	≤ 1

Table 5.2: Observed RV on SPEC OMP2001 benchmarks running with 16 threads and the `ref` data input

prevents them from remote memory accesses.

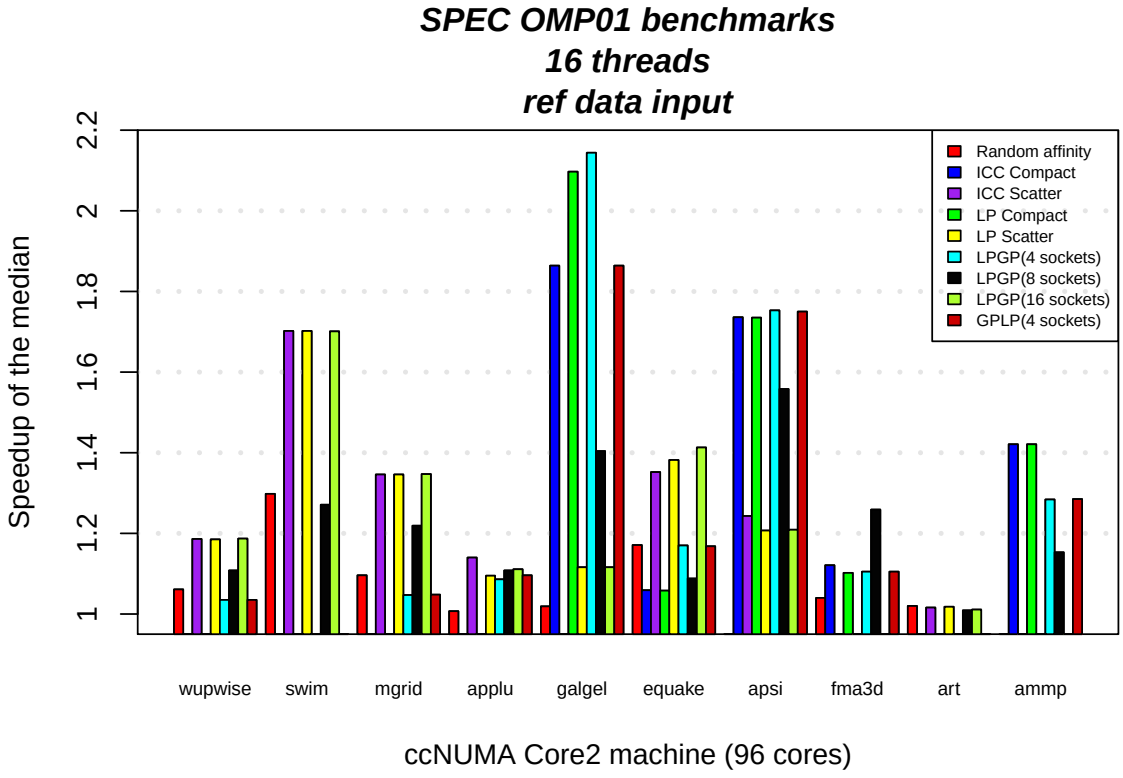


Figure 5.15: The observed speedups of the median execution times on the `ccNUMA` machine

In order to explain the reported speedups in Figure 5.15, we consider three benchmarks as case studies. We analyse the `swim`, `galgel` and the `fma3d` benchmarks, we selected them because we observe important speedups as far as the sharing and the non sharing of last level caches is concerned. Let us first start by the case of `swim`. This benchmark highlights the case where it is preferable to run each thread in a separate processor (`icc scatter`, `LP scatter` and

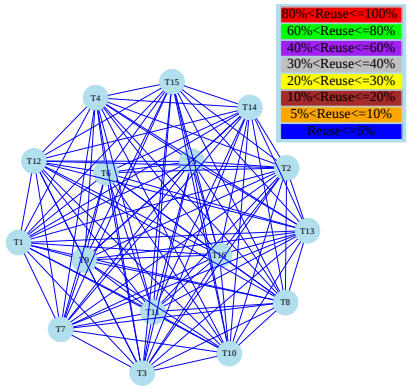


Figure 5.16: Affinity graph of `swim`

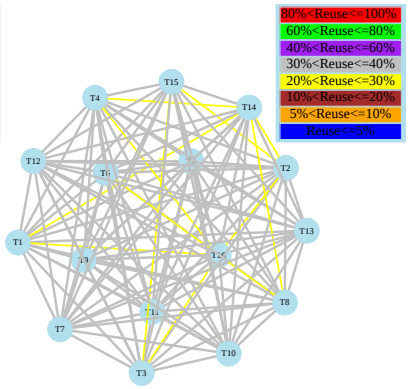


Figure 5.17: Affinity graph of `galgel`

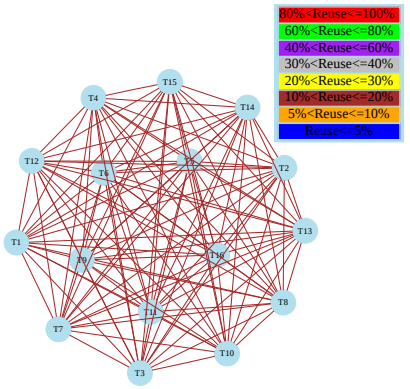


Figure 5.18: Affinity graph of `fma3d`

`LPGP(16 sockets)` strategies). Moreover, a wrong pinning can lead to a severe performance degradation in this machine. The important performance difference between the best and worst case in the `swim` benchmark can be explained mainly by two reasons. First, this benchmark does not exhibit significant inter-thread data reuse and sharing between its threads (See Figure 5.16). For this reason, it is preferable that each thread runs in a separate processor. Thus, it will have an exclusive access to a large L3 cache without contention from other threads. Second, distributing all the threads across the entire machine leads to a distributed memory allocation on the four memory nodes, this decreases the pressure from accessing one unique memory node, hence, a decrease in the memory latency access leading to better program performance.

The second case study is `galgel`. For this benchmark, program performance is better when all 16 threads are scheduled in a single node. We can see that the `icc compact`, `LP compact`, `LPGP(4 sockets)` and the `GPLP(4 sockets)` achieve the best program performance. It means cache sharing has a constructive performance effect on this benchmark. From our memory trace analysis, we can explain this performance behaviour as follows. First, regarding the amount of shared data in this benchmark (see Figure 5.17), we can say that it is significant. Second, the analysis of working sets shows that the sum of the individual working set of each thread is four times greater than the total working set of the whole application. This situation shows that in `galgel`, there is significant opportunities of data reuse and share which are concretely transformed into performance benefit. Besides, we can clearly observe that the application dependent strategies perform better than the application independent ones. This is predictable since the former are built upon a model which maximises the opportunities of data reuse and sharing.

Finally, we present the case of the `fma3d` benchmark. This benchmark is interesting in the sense that it highlights an intermediate performance situation between the use of a single compute node and using them all. Distributing all the threads across the entire machine does not improve performance. According to the Speedup-Test protocol, the `icc compact`, `LP compact`, `LPGP(4 sockets)` and `GPLP(4 sockets)` achieve a ≈ 1.12 speedup. We observe also that `LPGP(8 sockets)` strategy (pinned to 2 compute nodes) improves program performance by a 1.26 speedup. The study of the working sets of the application combined with the result of the data reuse metric shows that the `fma3d` is not exhibiting a significant inter-thread data reuse, but still, it is not negligible (see Figure 5.18). This may explain why running the sixteen threads in two nodes leads to better performance. It allows each pair of threads to run in a

separate processor, hence, take benefit from the large L3 cache and exploiting opportunities of data reuse. The memory allocation policy also, contributes to the performance of the `LPGP(8 sockets)` strategy. Due to the first touch policy, the memory allocation is distributed in the two nodes. This alleviate the problem of the memory access contention compared to the case where all the memory pages are allocated to a single node.

In order to check how the SPEC OMP2001 applications behave when all the cores of the `ccNUMA` machine are used, we run each application with 96 threads using the `ref` data input. We have to notice that we have not experimented the `Random` strategy. The analysis of program execution times shows that SPEC OMP01 applications exhibit an RV in the range of $15-44\%$ when thread affinity is not enabled. The performance results under this configuration are not surprising, as usual it is due to thread migration in one side and to the high impact of initial wrong pinning (in this case due to NUMA penalties). When the SPEC OMP applications are launched with thread affinity, the observed variability is less than 6% in almost all cases.

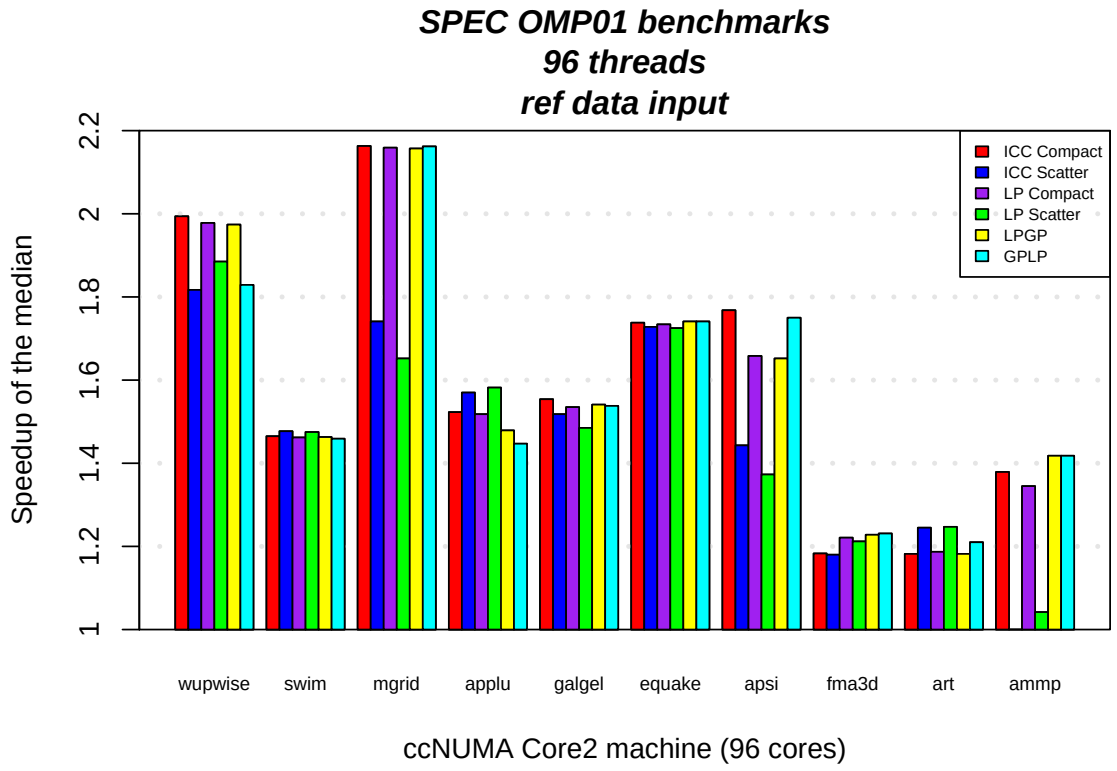


Figure 5.19: The observed speedups of the median execution times on the `ccNUMA` machine

As usual, we present performance results in terms of speedups (computed with the Speedup-Test protocol). Figure 5.19 report the observed speedups of the median execution times. We can conclude with the following observations:

1. We observe statistically significant speedups in almost all the tested benchmarks whatever the tested thread pinning strategy.

2. We do not observe any important performance difference between application independent and application dependent thread pinning strategies.

3. There are still some benchmarks where wrong pinning can lead to significant performance degradation. For instance, we observe an ≈ 0.72 slowdown between the `icc compact` and the `icc scatter` strategies in `ammp`.

4. Application dependent thread pinning strategies provide better results when the potential of data share in the parallel applications is high.

Synthesis with SPEC OMP2001 experimental results

Every SPEC OMP application has been executed 35 times on every machine, with different number of threads and on two different data sets (`train` and `ref`), according to nine thread pinning strategies. To summarise our experimental observations, we use three synthetic tables that reflect the speedups obtained through the thread pinning strategies with respect to the default `no affinity` strategy of the OS scheduler. Table 5.3 shows the overall sample speedup of every thread pinning strategy on the `Core2` and `Nehalem` SMP machines (having 8 cores each). We show the speedups of the average and the median execution times of all SPEC OMP applications executed with 4, 6 and 8 threads. Table 5.4 and Table 5.5 illustrate the same performance metrics on the HPC `ccNUMA` machine where the benchmarks have been executed with 16 and 96 threads. When only 16 threads are used (Table 5.4), the LPGP and GPLP strategies may have some variants which are the number of sockets used for executing 16 threads (described Table 5.4). In all the tables, we also report the minimal and maximal observed variances of the program execution times in order to study performance stability. Below we give our experimental conclusions and analyses:

1. On the `Core2` and `Nehalem` machines (8 cores), we can observe that fixing a thread affinity leads to marginal speedups and slowdowns (see Figures 5.13 and 5.14). This means that in terms of average or median execution times, letting the OS decide about thread placement is not a poor strategy. However, the performance variation is high (up to 82.24 for the `Core2` machine). Consequently, if performance stability is an additional quality criteria, it is better to fix thread affinity (check the maximal observed variances in Table 5.3 except for the random affinity strategy).

2. On the `ccNUMA` machine (Table. 5.4 and Table 5.5), we observe speedups for all thread affinity strategies (no slowdown). This means that using Linux thread scheduler is not a good choice in terms of performance.

3. When 96 threads are used on the `ccNUMA` machine (Table 5.5), the speedups are more significant. The reason is that the OS thread scheduler gives higher priority to work balancing compared to NUMA latencies: while the Linux kernel is able to distinguish between the latencies of distinct NUMA nodes, it still prefers to schedule threads to free available cores (to optimise work balancing by keeping all cores busy) even if such work balancing increases the cost of memory accesses (remote access to NUMA nodes). Consequently, a poor overall performance is observed if `no affinity` is fixed because some cores access data to remote memory nodes.

4. We do not observe any important difference, in terms of speedups, between application independent and application dependent strategies. This means that the price of profile guided methods is not easy to justify compared to cheap and easy-to-use `icc scatter` or `compact` strategies. One of the explanations is that fixing an affinity does not allow any thread migration during the execution of the application. Since any parallel application code may have different phases, it would be only by luck that the same thread placement

		Overall speedup (mean)		Overall speedup (median)		Min and max observed performance variances	
Thread pinning strategy		Core2 SMP	Nehalem SMP	Core2 SMP	Nehalem SMP	Core2 SMP	Nehalem SMP
Application independent	Random	0.995	0.995	0.992	1.001	[0.00 ; 82.24]	[0.00 ; 12.25]
	icc compact	0.952	0.995	0.944	0.996	[0.00 ; 0.22]	[0.00 ; 0.05]
	icc scatter	1.013	0.998	1.004	0.999	[0.00 ; 0.25]	[0.00 ; 0.31]
Application dependent	LP compact	0.952	0.995	0.945	0.996	[0.00 ; 0.11]	[0.00 ; 0.32]
	LP scatter	1.019	1.007	1.011	1.008	[0.00 ; 0.11]	[0.00 ; 0.27]
	LPGP	1.022	1.032	1.014	1.032	[0.00 ; 0.06]	[0.00 ; 0.04]
	GP	-	1.012	-	1.013	-	[0.00 ; 0.03]

Table 5.3: Overall sample speedups of the tested thread affinities with SPEC OMP2001 benchmarks running on the `Core2` and the `Nehalem` SMP machines. The baseline thread placement strategy is the OS free affinity. Each benchmark is executed repeatedly 35 times, using each run the `train` input dataset and with 4, 6, and 8 threads. The minimal and maximal observed performance variances of the OS free affinity are [0.01 ; 82.24] on `Core2` machine and [0.00; 0.52] on `Nehalem` machine.

Thread pinning strategy		Overall speedup (mean)	Overall speedup (median)	Min and max observed performance variances
Application independent	Random	1.053	1.046	[0.05 ; 3329.47]
	icc compact	1.025	1.018	[0.06 ;12.62]
	icc scatter	1.159	1.153	[0.08 ; 44.61]
Application dependent	LP compact	1.024	1.016	[0.08 ; 5.7]
	LP scatter	1.165	1.155	[0.10 ; 60.45]
	LPGP(4 sockets)	1.086	1.078	[0.03 ; 6.29]
	LPGP(8 sockets)	1.211	1.203	[0.01 ; 14.57]
	LPGP(16 sockets)	1.165	1.157	[0.02 ; 53.33]
	GPLP(4 sockets)	1.083	1.075	[0.05 ; 10.63]

Table 5.4: Overall sample speedups of the tested thread affinities with SPEC OMP2001 benchmarks running on the `ccNUMA` machine. The baseline thread placement strategy is the OS free affinity. Each benchmark is executed repeatedly 35 times, using for each run the `ref` input dataset and 16 threads. The minimal and maximal observed performance variances of the OS free affinity are [0.02 ; 12015.19]

gives the optimum for all phases. This favours to investigate other affinity solution based on thread migrations.

In the previous sections, we presented in details the result of the experiments related to the impact of the thread affinity on the program performance of SPEC OMP applications. The next section presents briefly our experimental results for the NBP benchmarks suite.

5.4.2 NAS Parallel Benchmarks

In order to not limit our performance study and analysis to SPEC OMP2001 benchmarks, we decided to extend our work to include the Nas Parallel Benchmarks (NPB). We follow the same experimental methodology and run each benchmark from the NBP suit under multiple thread pinning strategies. As in the case of SPEC OMP applications, the selected thread affinity strategies are split into application dependent and application independent strategies. We repeated the execution of each NPB application 35 times using the `B class` as data input on all the tested machines. The tested number of threads was 4, 6 and 8 on the SMP machines and 16

Thread pinning strategy		Overall speedup (mean)	Overall speedup (median)	Min and max observed performance variances
Application independent	icc compact	1.557	1.562	[0.11 ; 2.18]
	icc scatter	1.426	1.428	[0.20 ; 1.89]
Application dependent	LP compact	1.556	1.559	[0.17 ; 2.17]
	LP scatter	1.42	1.422	[0.10 ; 2.33]
	LPGP	1.565	1.568	[0.10 ; 3.85]
	GPLP	1.566	1.569	[0.14 ; 3.99]

Table 5.5: Overall sample speedups of the tested thread affinities with SPEC OMP2001 benchmarks running on the `ccNUMA` machine. The baseline thread placement strategy is the OS free affinity. Each benchmark is executed repeatedly 35 times, using for each run the `ref` input dataset and 96 threads. The minimal and maximal observed performance variances of the OS free affinity are [28.13 ; 3364.08]

threads on the `ccNUMA` machine[5].

Running the OpenMP NPB applications without thread affinity leads to a relative variability of program execution times in the range of $0-8\%$ on the `Nehalem` SMP machine and in the range of $0-34\%$ on the `Core2` SMP machine. On the other hand when thread affinity is enabled, we observed a relative variability in the range of $0-6\%$ in most cases on both machines. Furthermore, we observed that program performance is more stable on the `Nehalem` machine than on the `Core2` machine. Regarding the performance of the NPB applications on the `ccNUMA` machine, we observed a performance variability in the range of $8-30\%$ when the threads are launched without affinity. Running the NPB applications with thread affinity enabled on the `ccNUMA` machine leads to a performance variability less than 6% in most cases.

As we did for SPEC OMP01 results, we make a synthesis of performance results in two tables to reflect the speedups of thread pinning strategies compared to the `no affinity` configuration. Table 5.6 shows the overall sample speedup of every thread pinning strategy on the `Core2` and `Nehalem` SMP machines. Like the computed overall speedup in SPEC OMP applications, we report the speedups of the average and the median execution times of the tested NPB applications. Table 5.7 shows the overall sample speedup on the `ccNUMA` machine. This table contains some variants of the `LPGP` strategy which are the number of sockets used for executing the 16 threads. From the analysis of the two tables we conclude with the following:

1. On the `Core2` and `Nehalem` machine, we can observe that fixing a thread affinity leads to marginal speedups and slowdowns. This observation confirms the results obtained with the SPEC OMP2001 applications. In these two machines, letting the operating system decide about the placement of threads leads to acceptable average or median execution times. However, the observed RV in this case is significant.

2. On the `ccNUMA` machine, we observe speedups for approximately all the tested thread affinity strategies. This means that not fixing the affinity impacts negatively the performance of NPB applications. This observation is not true for the `LP compact` and the `icc compact` strategies. Indeed, we observe slowdowns with these two thread affinity strategies.

3. The poor behaviour of the `LP compact` and `icc compact` strategies on the `ccNUMA` machine is related to the nature of the NPB applications. In almost all the applications,

[5]We did not have the opportunity to test the case of 96 threads on the `ccNUMA` machine because of time access restriction.

		Overall speedup (mean execution times)		Overall speedup (median execution times		Min and max observed performance variances	
Thread pinning strategy		Core2 SMP	Nehalem SMP	Core2 SMP	Nehalem SMP	Core2 SMP	Nehalem SMP
Application independent	Random	0.984	0.968	0.962	0.996	[0.00;569.00]	[0.00;26.35]
	icc compact	0.869	0.879	0.858	0.879	[0.00;11.16]	[0.00;0.19]
	icc scatter	0.969	0.999	0.957	0.999	[0.00;5.43]	[0.00;0.06]
Application dependent	LP compact	0.87	0.88	0.859	0.879	[0.00;10.05]	[0.00;0.28]
	LP scatter	0.97	1.001	0.957	1	[0.00;5.11]	[0.00;0.08]
	LPGP	1.052	1.003	1.043	1.003	[0.00;6.73]	[0.00;0.06]
	GP	-	1.001	-	1	-	[0.00;0.09]

Table 5.6: Overall sample speedups of the tested thread affinities with Nas Parallel benchmarks running on the `Core2` and the `Nehalem` SMP machines. The baseline thread placement strategy is the OS free affinity. The used data input was `class B`. The numbers of threads were 4, 6 and 8 for every benchmark. The number of repetitive runs per benchmark was 35. The minimal and maximal observed performance variances of the OS free affinity are [0.00 ; 602.341] on `Core2` machine and [0.00; 26.35] on `Nehalem` machine.

Thread pinning strategy		Overall speedup (mean execution times)	Overall speedup (median execution times	Min and max observed performance variances
Application independent	icc compact	0.924	0.927	[0.00;0.35]
	icc scatter	1.289	1.292	[0.00;2.22]
Application dependent	LP compact	0.925	0.927	[0.00;0.15]
	LP scatter	1.298	1.301	[0.00;1.72]
	LPGP(4 sockets)	1.01	1.011	[0.00;0.24]
	LPGP(8 sockets)	1.2	1.202	[0.00;0.36]
	LPGP(16 sockets)	1.295	1.296	[0.00;0.16]
	GPLP(4 sockets)	1.014	1.015	[0.00;0.22]
	GPLP(4 sockets)	1.014	1.015	[0.00;0.22]

Table 5.7: Overall sample speedups of the tested thread affinities with Nas Parallel benchmarks running on the `ccNUMA` machine. The baseline thread placement strategy is the OS free affinity. The used data input was `class B`. The numbers of threads were 16 for every benchmark. The number of repetitive runs per benchmark was 35. The minimal and maximal observed performance variances of the OS free affinity are [0.00 ; 52.24].

the analysis of the memory traces shows a small inter-thread data sharing. Since these `compact` strategies place the 16 threads in one compute node, the contention on the shared buses and caches leads to a performance degradation compared to strategies which distribute the threads among all the available compute nodes and sockets.

4. Due to the small amount of inter-thread data sharing, it makes no difference, in terms of performance between the application dependent and application independent strategies. Yet, we observe that application dependent strategies are slightly better than application independent ones.

5.5 Conclusion

We investigate various application-wide cache-aware thread pinning strategies for SPEC and NPB OpenMP applications. We performed a statistical performance evaluation and analysis and demonstrated that fixing an affinity provides statistically significant performance improvements compared to the Linux OS strategy. However, the performance improvement is marginal on UMA `Core2` and `Nehalem` machines, but the performance stability is better. On the tested

`ccNUMA` machine, the speedups of all thread pinnings are significant because the OS thread scheduler gives higher priority to work balancing among cores against NUMA sensitive scheduling.

Interestingly enough, we demonstrated that application independent strategies (`icc scatter` and `icc compact`) provide equivalent performance gains compared to profile guided (application dependent) methods. The later observation does not suggest that the profile guided methods are useless or inefficient. First, we have shown that since application dependent strategies are more cache-aware, they provide better performance enhancement for multi-threaded applications which implement an important amount of data sharing. Second, we have investigated only cache effects, while there are other factors that have to be considered in order to compute effective thread pinning strategies. For example, it is possible also to consider bus contention, prefetch contention, last level cache contention and memory controller contention.

Another explanation for the weak observed speedups can be due to modelling issues. Since the *affinity graph* is built as an aggregated view of the whole execution, it does not account for temporally distinct sharing patterns in the application. In other words, although, the *affinity graph* reports data sharing, we are not really sure that this data sharing is effectively transformed into performance improvement by thread affinity. To overcome this limitation, we suggest to reduce the granularity of our profile-guide method. We think that profile guided methods should be improved by considering program phases to decide variable thread pinnings (migration).

In the next chapter, we investigate the performance impact of thread pinning and migration strategies per parallel region for SPEC OMP and NPB applications.

Chapter 6

Dynamic Thread Pinning for Phase-Based OpenMP Programs

This chapter presents an approach that extends the profile guided method seen in the previous chapter. In fact, instead of computing an application-wide thread affinity strategy, a distinct thread affinity is computed for each OpenMP parallel region. This allows to change thread affinity dynamically (thread migrations) between parallel regions at runtime.

6.1 Introduction

In Chapter 5, we showed that thread affinity is an important factor that has to be considered when it comes to accelerate program execution times. Another advantage of fixing thread affinity is better performance stability. We also showed that while fixing thread affinity during the whole execution provides statistically significant performance improvements in NUMA machines, the obtained speedups are negligible on SMP machines. Nevertheless, we think that there is still some potential to further enhance the performance gain using thread affinity by exploiting phase-based behaviour in OpenMP programs.

In this chapter we are going to study a phase-based or a dynamic thread pinning technique. The later rely on the *control flow graph* of a parallel execution in a given program. A *basic block* in this *control flow graph* of a parallel execution can be defined using different granularities: a sequence of some instructions, a function call, etc. Most often, OpenMP programs implement multiple parallel regions which are called multiple times in an iterative way. In the context of our work, we consider the *control flow* as a graph representing a sequence of calls to distinct parallel regions in a OpenMP program. This also means that we define an OpenMP phase as the event of executing a parallel OpenMP region. Since, we are dealing with thread affinity, and in order to reduce the number of thread migrations, we think that the parallel region granularity represents a good trade-off between better accuracy and lower overhead.

The remainder of this chapter is structured as follows. We first present in Section 6.2 a synthetic example aiming to show the effectiveness of using dynamic per-parallel regions thread affinity within OpenMP programs. Section 6.3 describes the methodology that we use in order to compute a distinct thread affinity for each parallel region. In Section 6.4, we present the details of the experimental setup. Finally, we give in Section 6.5 the experimental results and analysis, then we conclude.

6.2 Motivation and problem description

In the previous chapter, we showed that fixing thread affinity leads to better performance stability. We also showed that while fixing thread affinity during the whole execution provides statistically significant performance improvements in NUMA machines, the obtained speedups are negligible on SMP machines (Chapter 5). However, we think that there is still some potential to further enhance the performance gain using thread affinity by exploiting phase-based behaviour in OpenMP programs.

In our study, we define an OpenMP phase as a unique and a distinct OpenMP parallel region. In OpenMP, each structured code started by the construct `#pragma omp parallel` in C/C++ or `!$omp parallel` in Fortran is a new parallel region. Listing 6.1 shows an example of a program with two parallel regions. These regions translate into two distinct OpenMP phases.

Listing 6.1: Two OpenMP parallel phases program

```
#pragma omp parallel
{
  .....
}

#pragma omp parallel
{
  .....
}
```

To illustrate the benefit of changing thread pinning between consecutive OpenMP parallel regions, we use a synthetic micro-benchmark implementing two parallel regions running with 4 threads. The implemented sharing behaviour is as follows:

- In the first parallel region:
  - Threads 1 and 2 share some data.
  - Threads 3 and 4 share some data.
- In the second parallel region:
  - Threads 1 and 3 share some data.
  - Threads 2 and 4 share some data.

We run the micro-benchmark multiple times and using multiple thread pinnings on top of an Intel SMP[1] with two quad-core processors, each couple of cores share an L2 cache[2]. We consider the `no affinity` strategy as the base comparison configuration. Figure 6.1 reports the speedups of the median execution times of the tested pinning techniques. In total we have six thread pinning strategies. Except one configuration where the pairs of threads were placed on a single socket, we place each pair of threads on cores sharing an L2 cache in a distinct socket. For instance, if we consider from the figure, the case of the configuration `C2: 1324,1234:two sockets`, means that we place in phase 1, the pair of threads (1,3) in socket #1 and the pair (2,4) in socket #2. Similarly, we place in phase 2, the pair of threads (1,2) in socket #1 and the pair (3,4) in socket #2.

[1] There are no NUMA effects, the machine has a single memory domain.
[2] More details about the machine can be found in page 100

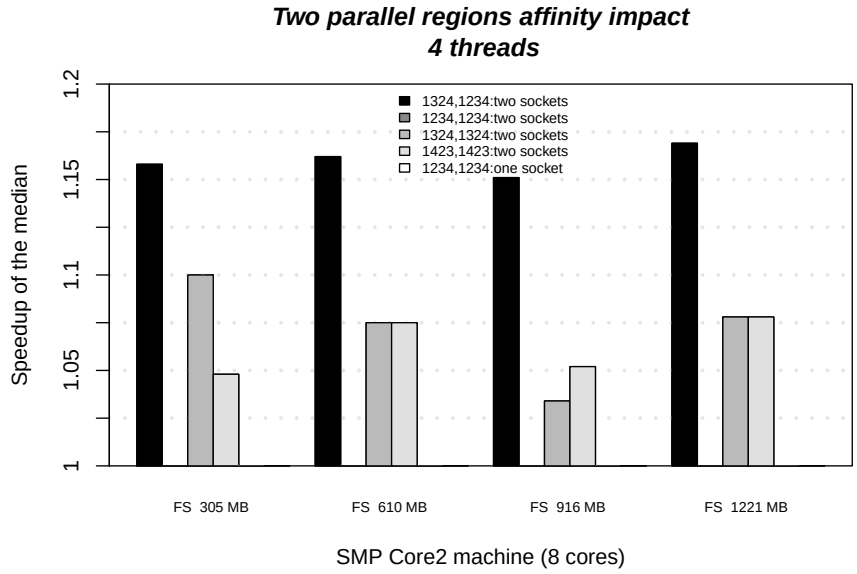


Figure 6.1: `Core2` SMP machine results (Only statistically significant speedups are plotted).

From this simple experiment, we can conclude that changing the affinity between OpenMP parallel phases is beneficial and can lead to non negligible performance improvement over a fixed affinity for the whole program or a `no affinity` strategy. We conducted a memory trace analysis for this program, we computed an affinity graph for each of its parallel phases. Figure 6.2 shows the graphical view of the affinity graph of each parallel region. The width and the color of each edge in the graph is proportional to the amount of data sharing between each pair of threads compared to the total number of memory accesses of the whole application[3]. For instance, edges with a red color represent high amount of inter-thread data sharing in the application. For this micro-benchmark, the memory trace analysis confirms the results shown above. If we consider the case of the first parallel region, achieving good performance requires that thread 1 has to be close to thread 2. Similarly thread 3 has to be close to thread 4.

In this section we have shown the importance of dynamic thread pinning per parallel phases for a simple program. In the next section, we explain the methodology we follow to compute a thread pinning per parallel regions and check the effectiveness of this approach for SPEC OMP2001 and NPB programs and for some other synthetic benchmarks.

6.3 Parallel OpenMP phases extraction and thread pinning

Once again, we focus on data sharing in order to compute effective thread pinning strategies. On the contrary to what we did when computing an *application-wide* thread affinity, this time we compute a thread pinning for each parallel region in the OpenMP program. Our extended profile guided method consists of multiples steps. First, we instrument OpenMP constructs to add function calls allowing us to know entry and exit point of OpenMP parallel regions. Second, after determining entry/exit point for each parallel region, it is possible to collect a distinct memory trace profile for each of them. Third, for each distinct memory trace profile, we build an affinity graph. This means that we build a distinct affinity graph for each parallel

[3]All the accesses of all threads in the application.

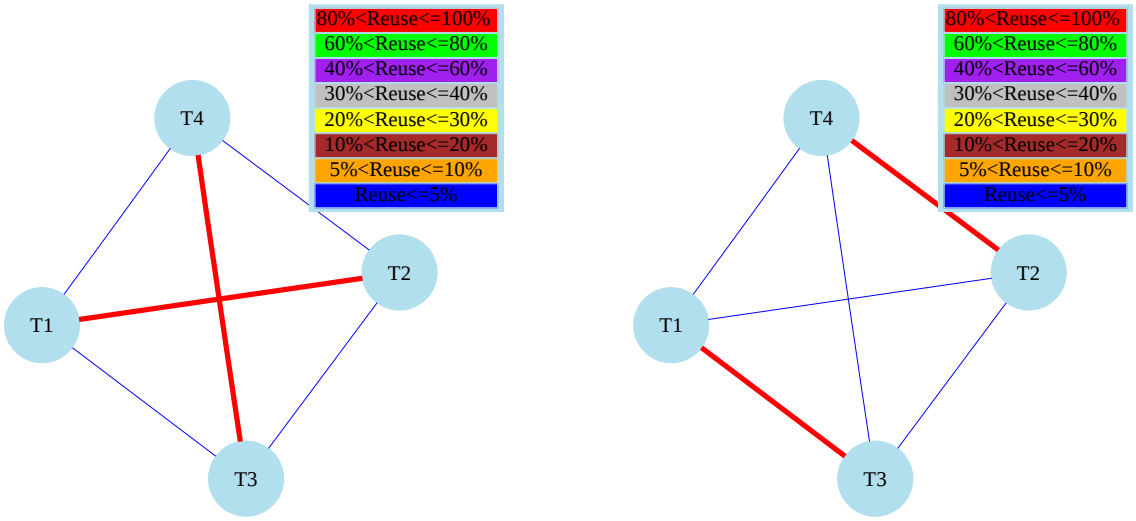


Figure 6.2: Affinity graphs of the micro-benchmark

Listing 6.2: Before OPARI translation

```
#pragma omp parallel
{
   printf("Hello world\n");
}
```

Listing 6.3: After OPARI translation

```
POMP_Parallel_fork(d)
#pragma omp parallel
{
   POMP_Parallel_enter(d)
   printf("Hello world\n");
   POMP_Parallel_end(d)
}
POMP_Parallel_join(d)
```

region. Finally, we apply multiple partitioning techniques in order to decompose each distinct affinity graph into multiple parts and hence, we can compute a thread affinity for each parallel region. We detail in the following sections all these steps.

6.3.1 Automatic detection of OpenMP parallel regions

Regarding OpenMP programs, computing a thread affinity for a parallel region requires to detect the entry and exit events of that region. All the events are detected using the `OPARI` [MMSW02] instrumentation tool. `OPARI` is a component of a global framework for parallel program performance evaluation and measurement. The objective of `OPARI` is to provide a performance and measurement interface for OpenMP. It is a source-to-source translation tool which automatically adds function calls to a `POMP` runtime measurement library. This library is used to collect runtime performance data for OpenMP applications. `OPARI` supports C/C++ and Fortran programming languages. The idea behind the concept is to detect each OpenMP pragma/directive and add functions calls to the `POMP` library. This method allows us to be compiler and runtime independent. In our approach, we do not use the `POMP` library for performance measurement. Instead, we have made changes in order to achieve dynamic thread pinning for each parallel region. Listings 6.2 and 6.3 present an example of how the translation process is achieved.

The translation process is done as follows:

- `OPARI` translates OpenMP pragmas to function calls of the form `POMP_Name_type(d)`
  - `Name` refers to name of the OpenMP directive
  - `type` is either fork, join, enter, exit, begin, or end
  - `d` is the context descriptor of the OpenMP directive
- The OpenMP program is linked with the `POMP` performance measurement library
- We modified the `OPARI` tool to expose only events related to parallel regions (enter/exit)

6.3.2 Memory trace profile and analysis for OpenMP regions

After the `OPARI` instrumentation, we make a memory tracing of the OpenMP application using the `PIN` [LCM+05] instrumentation framework. To account for multiple parallel regions (PR), and following the idea presented in in Section 5.2.2.1, we extended our C++ Pin tool. Unlike the application-wide memory trace collection, we associate to each thread $\|\mathcal{P}\|$ hash tables, where $\mathcal{P}$ represents the set of distinct parallel OpenMP regions in the application. Again, a hash table holds all the memory references accessed by a given thread. The entries of a hash table store the block identifier (BID) of the given memory reference (transformed to data blocks of 64 bytes size) and the number of accesses to this BID where the number of memory reads and writes are kept separate. From memory tracing, we build an *affinity graph* for each PR in the OpenMP application. In addition, we are able to deduce the parallel regions control flow graph `PRCFG`. It is a directed valued graph where the vertices represent the distinct PRs of the program and the edges represent the predecessor and the successor relationship between them. As reported before, an edge between a PR_i and PR_j is valued by the number of times the execution of the PR_i is followed by the execution of PR_j. Figure 6.3 shows an example of a parallel regions control flow graph of the `CG` benchmark from the NPB suit.

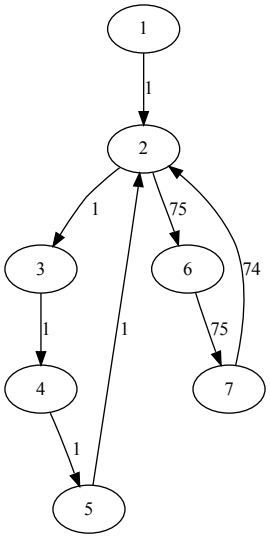


Figure 6.3: A parallel regions control flow graph of the `CG` benchmark

The next sections shows how we use memory trace information to build the *affinity graph*.

6.3.3 Building an affinity graph for each parallel region

The collected memory trace profile is used to build an *affinity graph* for each parallel parallel region in the program. Each *affinity graph* in the application is an undirected valued graph $G_p = (\mathcal{V}, \mathcal{E}, \alpha) \quad \forall p \in \mathcal{P}$. $\mathcal{V}$ is the set of application threads, $\mathcal{E} = \mathcal{V} \times \mathcal{V}$, $\alpha : \mathcal{E} \mapsto \mathbb{N}$ is a gain function applied to every pair of threads and $\mathcal{P}$ is the set of parallel regions implemented in the application. An *affinity graph* is a complete graph: we consider a fixed number of threads, then the number of edges in the graph is equal to $\frac{\|\mathcal{V}\| \times (\|\mathcal{V}\|-1)}{2}$.

The gain function $\alpha(T_i, T_j)$ represents the number of common accesses to common memory caches lines, accessed by both the T_i and T_j threads for a given parallel region. Besides the gain function we defined in the previous chapter that we call a simple model or SM, we added another metric to account for inter-thread data sharing, and we call it the read/write model or RWM. We added this model because we consider that it is important to separate read and write accesses from the performance perspective. The reason for that is we consider that a shared region of data wherein accesses are dominated by reads will have less impact on performance than a shared region of data wherein the read and write accesses are balanced. In fact, when the shared data are accessed only in a read mode, duplicating these data on multiple caches may not harm the performance in a great extent.

Let us precisely define α for an application with a fixed number of threads $n = \|\mathcal{V}\|$ and for a given parallel region $p \in \mathcal{P}$. The collected memory trace profile contains the information $A_p(T_i, b)$ which is the number of accesses of thread T_i to data block b at parallel region p. However, since we distinguish between reads and writes, then we exactly have $RD_p(T_i, b)$ and $WR_p(T_i, b)$ which is the number of reads and writes respectively performed by thread T_i to data block b and where $A_p(T_i, b) = RD_p(T_i, b) + WR_p(T_i, b)$. If we consider $B^p_{i,j}$ as the set of all data blocks accessed by the pair of thread (T_i, T_j) at parallel region p, then Equation 6.1 defines the function $\alpha(T_i, T_j)$, which is exactly the number of accesses to common memory blocks by both the threads T_i and T_j (with $T_i \neq T_j$):

$$\begin{aligned} \alpha(T_i, T_j) = \sum_{b \in B^p_{i,j}} & \min\left(RD_p(T_i, b), WR_p(T_j, b)\right) + \\ & \min\left(WR_p(T_i, b), RD_p(T_j, b)\right) + \\ & \min\left(WR_p(T_i, b), WR_p(T_j, b)\right) \end{aligned} \tag{6.1}$$

The next section shows the different tested thread pinning strategies that we compute.

6.3.4 Tested thread pinning techniques

Once all the affinity graphs are constructed for an application and for a given number of threads, we can use them to investigate multiple thread pinning strategies. Using graph partitioning techniques presented in Section 5.2.2.3, we decompose all affinity graphs into multiple disjoint partitions. The thread affinity of the application at each parallel region is computed by assigning each distinct graph partition to one shared cache in the multicore platform.

The overall speedup analysis performed in the previous chapter showed us that from the performance perspective, almost all the application dependent strategies provide similar program performance. For this reason, we limit our evaluation only for the following strategies (regard-

Listing 6.4: After OPARI translation

```
POMP_Parallel_fork(d)
#pragma omp parallel
{
  POMP_Parallel_enter(d)
  ...
  ...
  ...
  POMP_Parallel_end(d)
}
POMP_Parallel_join(d)
```

ing k-partitioning techniques), corresponding to the application of heuristics for solving graph k-partitioning problems at each level of the memory cache hierarchy of the parallel machine:

1. `LPGP` strategy. After an initial step of optimal computation of thread pairs, we proceed by a graph k-partitioning [KK98c]. It is a hierarchical strategy, where threads are first paired and pinned on shared L2 or L3 cache then thread pairs are partitioned and placed on the different sockets according to their affinity.

2. `GPLP` strategy. It is a hierarchical strategy. It starts by an initial graph k-partitioning to fix threads on sockets, then perform an optimal polynomial algorithm to compute thread pairs sharing L2 or L3 cache levels.

In addition to the application dependent strategies presented above, we consider in our evaluation, the following application independent strategies:

1. Run the application without affinity (`no affinity`).

2. Run the application with a `compact` strategy of the `icc` compiler (`icc compact`).

3. Run the application with a `scatter` strategy of the `icc` compiler (`icc scatter`).

6.3.5 Setting a per-parallel OpenMP thread pinning

If we consider an OpenMP program with multiple parallel regions, each thread in the application may have a different pinning for each parallel region highlighting different sharing patterns. The goal is to associate for each thread, the appropriate core number whatever the parallel region. To do so, we extended the `POMP` library by managing a global table where the number of rows is equal to the number of parallel phases and the number of columns is equal to the number of threads. In other words, this table associates a core number for each thread, and for a given parallel region, the value at the i_{th} row and the j_{th} column means the pinning of thread j at the parallel region i.

One may ask the following question: regarding the instrumented code, where is the appropriate location to add the `sched_set_affinity` function to set thread affinity? When we instrument the OpenMP code with `OPARI`, each time the parallel construct `#pragma omp parallel` is found, calls to the `POMP_Parallel_fork` and `POMP_Parallel_enter` functions (from the `POMP` library) are added to the instrumented code. As illustrated in Listing 6.4, while the function `POMP_Parallel_fork` is executed only by the master thread, `POMP_Parallel_enter` is executed

by each thread in the application. Consequently, we think that it makes sense to consider the effective thread pinning at the level of this function. Using some context information (allowing the library to know the current parallel region), and the global table of pinnings, each thread can independently set the affinity to core corresponding to the current parallel region.

Since we know where to set the thread affinity of each thread for each parallel region, we actually need to provide all the pinnings to the application before the beginning of the execution. In our modified version of the `POMP` library, we consider a solution with an environment variable `CPU_AFFINITY`. This variable contains all or a subset of thread pinnings corresponding to all or a subset of parallel regions separated by commas. Each pinning has colon-separated two parts. The first part indicates the number of the parallel region to which the pinning refers to. The second part is a space-separated list of CPUs or cores. Let us consider the following example for an application with two parallel regions: `export CPU_AFFINITY="1:0 1 2 3 4 5 6 7, 2:0 4 1 5 2 6 3 7"`. Each time the application reaches the parallel region one or two, the corresponding pinning is set accordingly. In the later example for instance, when the threads reach the parallel region number 1, thread 0 is pinned to core 0, thread 1 is to core 1, thread 2 to core 2 and so on. Similarly in parallel region number 2, thread 1 is pinned to core 0, thread 1 is pinned to core 4 and so on.

The next section shows our experimental setup before presenting the results of our performance measurement evaluation.

6.4 Experimental setup and methodology

This section describes the experimental setup that has been used to conduct the study. We describe the benchmarks, the hardware platforms and the experimental methodology.

6.4.1 Software environment

Our experiments have been conducted using various applications, some are micro-benchmarks and others are standard benchmarks. The former studies the necessary conditions to make the per parallel regions thread affinity effective to achieve good performance. The later benchmarks are used to study the performance of our approach in a general case. We run each application with 8 and 16 threads according to the number of available number of cores. We tested multiple thread placement strategies for every application, thread number, while repeating the execution 35 times (for statistical significance analysis). The benchmarks have been compiled using Intel compiler (`icc` 11.1) with flag `-O3 -openmp`. The description of the tested applications is given in the following sections.

A synthetic benchmark

This benchmark implements two OpenMP parallel regions, each with a distinct sharing pattern. The benchmark uses a single large rectangular (the width is much greater than the height) matrix which is subdivided into equal parts among all the intervening threads. Figure 6.4 reports the inter-thread sharing structure of the benchmark. For example, in parallel region 1, data sharing is between (T_1, T_5), (T_2, T_6), (T_3, T_7) and (T_4, T_8) thread pairs. Similarly, in parallel region 2, data sharing is between (T_1, T_2), (T_3, T_4), (T_5, T_6) and (T_7, T_8) thread pairs. Cache lines sharing between threads is implemented by allowing for each pair of threads to access common cells from the portion of the array that has been assigned to them. For each

assigned portion from the array, each thread performs simple computations like additions and multiplications.

Figure 6.4: The two inter-thread data sharing patterns for the two distinct parallel regions of the benchmark.

A matrix multiply benchmark

Matrix multiplication is often used as a benchmark to evaluate the performance of memory optimisation techniques. It is also an important routine in many optimised linear algebra libraries. In this section, we examine the program performance of two successive matrix multiplications. Each matrix multiplication implements a distinct sharing pattern. The computation is of the form $A \times B + B \times A = C$, where A, B and C are matrices of equal sizes. We can find this form of matrix computations in linear systems like the Sylverster equation [Syl84], Lyapunov equation [BS72] or Algebraic Riccati equation [LR95]. In order to implement two distinct inter-thread data sharing patterns, we consider that while A and C are dense matrices, matrix B has a special structure. In fact, B is decomposed into a 4 blocks where the blocks in the diagonal have non zero values and the remaining blocks have zero values.

Figure 6.5 shows the matrices structure and work distribution for 16 threads performing two successive matrix multiplications. For each matrix multiply computation, each thread computes a single and unique block in matrix C. Computing the values of a block in matrix C requires to access to a block in matrix A and a block in matrix B. With this work distribution and considering the special structure of matrix B, we can observe that the four non-zero blocks $(B1, B6, B11, B16)$ in B are shared by distinct groups of threads. We can also observe that the inter-thread blocks sharing is different between these two successive computations. For instance, $B1$ is shared between threads $T1, T5, T9, T13$ in phase one, and between threads $T1, T2, T3, T4$ in phase two.

SPEC OMP01 and NPB benchmarks

Besides the presented micro-benchmarks, we conducted also experiments using all SPEC OMP01 [Sta06] and NAS Parallel Benchmarks (NPB) [JFY99]. Regarding these benchmarks, we used the `ref` data input with SPEC OMP01 and the `Class B` with NPB benchmarks.

6.4.2 Hardware setup

We conducted all our experiments on four platforms:

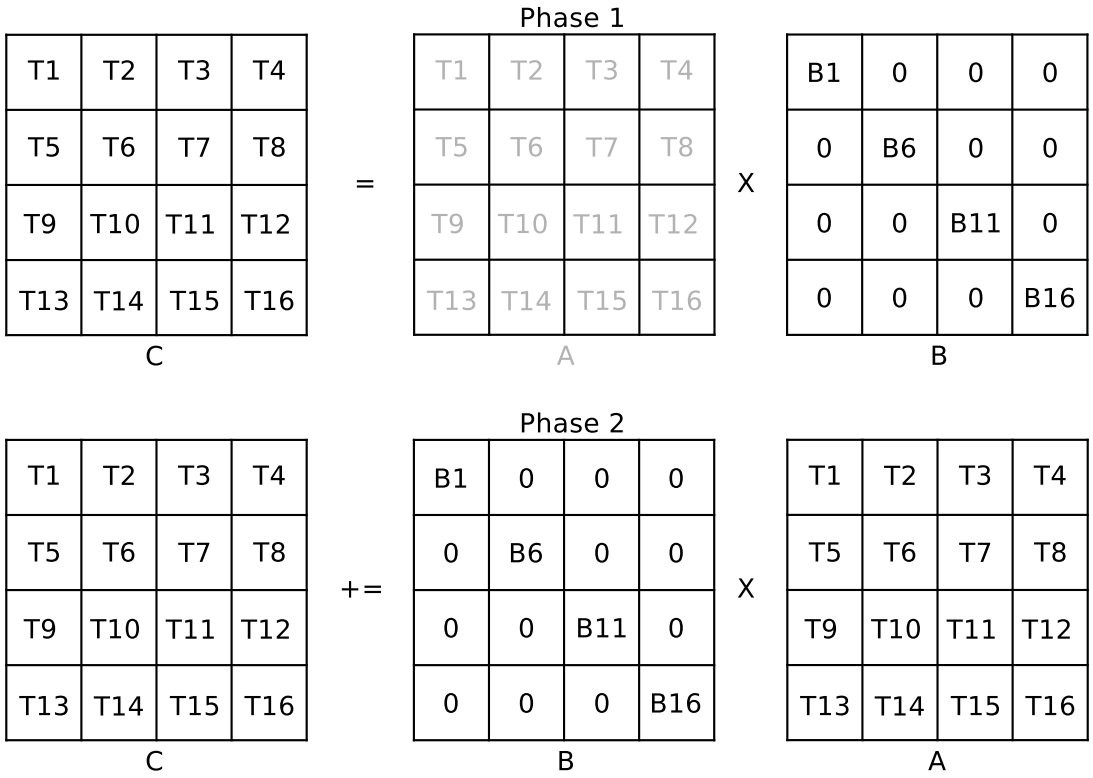


Figure 6.5: Matrices structure of a two successive parallel matrix multiply. Each thread accesses to its own block in matrices A and C. Threads with the same color means that they share the same block in matrix B.

1. The `Core2` (8 cores) machine that we name also the SMP machine (Figure 5.10). It is a single SMP machine with two `Clovertown` sockets (Intel Xeon E5345 with the `Core2` micro-architecture). Each processor has 4 cores, each pair of cores have a shared L2 cache. The platform has two L2 4MB caches per socket, for both instructions and data. The core frequency is 2.33 GHz. Each core has a separate 32KB L1 data and instruction caches. The main memory size is 4 GB RAM. The front-side bus has a clock rate of 1.33 GHz.

2. The `Nehalem` (8 cores) machine (Figure 6.6). It is an Intel `NUMA` machine with two compute nodes. Each NUMA domain has a single `Gainestown` processor (Intel Xeon X5570 with the `Nehalem` micro-architecture). Each processor has 4 cores (2 threads per core) with an inclusive shared L3 cache. So in total, we have 16 hardware threads. The platform has two L3 caches of 8 MB, one on each chip, for both instructions and data. The core frequency is 2.93 GHz. Each core has a private 256KB L2 cache unified for data and instructions. In addition, each core has a separate L1 data and instructions caches with 32 KB each. The main memory size is 24 GB (12 GB in each NUMA domain). Each chip in the platform has an integrated memory controller. Communication between sockets is achieved through the `QPI` (Quick-Path Interconnect).

3. The `Shanghai` (8 cores) machine (Figure 6.7). It is an AMD `NUMA` machine with two compute nodes. Each NUMA domain has a single `Opteron` processor (AMD 2378 with the `K10` micro-architecture). Each processor has 4 cores with an exclusive shared L3 cache. The platform has two L3 caches of 6 MB, one on each chip, for both instructions and data. The core frequency is 2.4 GHz. Each core has a private 512KB L2 cache unified for data and instructions. In addition, each core has a separate L1 data and instructions

caches with 64KB each. The main memory size is 32 GB (16 GB in each NUMA domain). Each chip in the platform has an integrated memory controller. Communication between sockets is achieved through the `HT` (Hyper-Transport) protocol.

4. The `Barcelona` (16 cores) machine (Figure 6.8). It is an AMD `NUMA` machine with four compute nodes. Each NUMA domain has a single `Opteron` processor (AMD 8347HE with the `K10` micro-architecture). Each processor has 4 cores with an exclusive shared L3 cache. The platform has four L3 caches of 2 MB, one on each chip, for both instructions and data. The core frequency is 1.9 GHz. Each core has a private 512KB L2 cache unified for data and instructions. In addition, each core has a separate L1 data and instructions caches with 64KB each. The main memory size is 32 GB (8192 B in each NUMA domain). Each chip in the platform has an integrated memory controller. Communication between sockets is achieved through the `HT` (Hyper-Transport) protocol.

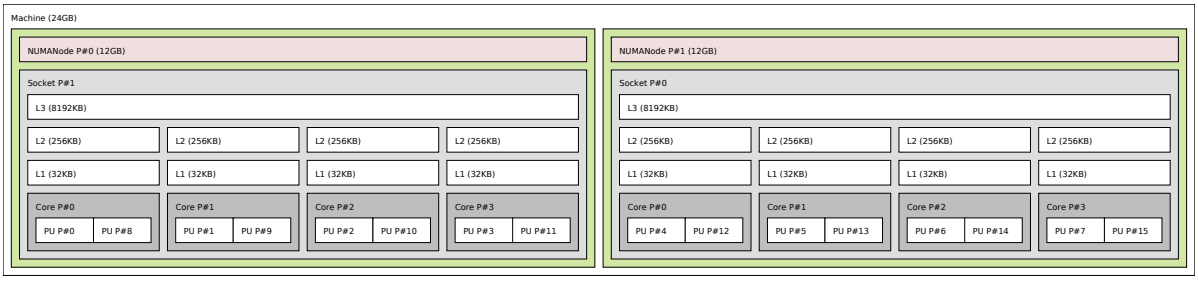


Figure 6.6: `Nehalem` NUMA machine architecture

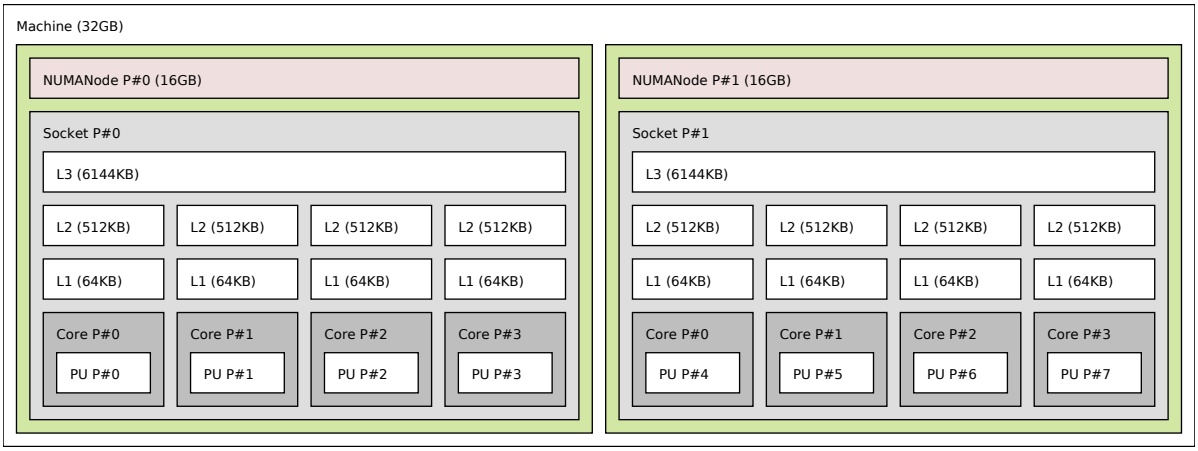


Figure 6.7: `Shanghai` NUMA machine architecture

6.4.3 Evaluation methodology

There are multiple programs where it is not necessary to fix thread affinity for all the parallel regions. In general, we can consider that we have a range between 1 and 5 parallel regions which dominate the total execution time and where the execution times of the remaining parallel regions can be considered as negligible. Focusing only on these *hot parallel regions* may lower the frequency of thread migrations, and consequently improve performance. To do so, we run each benchmark natively with the desired data input, and we measure the accumulated execution time of each parallel region in the OpenMP program. After having the execution

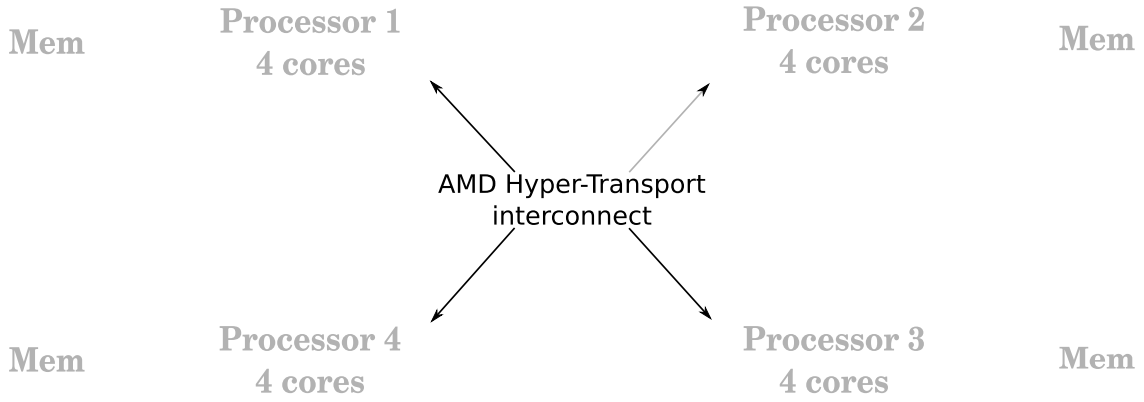


Figure 6.8: `Barcelona` NUMA machine architecture

times of all the parallel regions, we perform a sort in a descending order of execution times. Finally, we consider the first parallel regions which contribute to $\approx 90\%$ of the total execution time.

Sometimes, the OpenMP program performance is dominated by the performance of a unique parallel region. For these programs, we fix thread affinity only for the dominating parallel region. Experimentally, we define the threshold for such parallel regions to be at least 70% from the total execution time of the whole application. We do such a choice because we consider that the expected performance enhancement from fixing thread affinity for the remaining parallel regions may be marginal compared to the potential performance degradation.

6.5 Experimental evaluation of phase-based thread pinning

This section introduces a performance evaluation and analysis of the effectiveness of the per parallel regions thread affinity strategy with some micro-benchmarks, SPEC OMP01 and NPB benchmarks.

6.5.1 Performance analysis using micro-benchmarks

Let us start our performance evaluation and analysis with some synthetic benchmarks from the computation granularity and the amount of data sharing perspectives. We performed the experiments presented in this section using the `Intel Core2` SMP and the `Intel Nehalem` NUMA machines.

6.5.1.1 Synthetic benchmark with two inter-thread data sharing patterns

We start the evaluation of the effectiveness of changing thread affinity through different parallel regions with a simple synthetic benchmark.

Case study 1: each pair of threads share 100% of their assigned data
Figures 6.9 and 6.10 report the observed program performance of the benchmark running with 8 threads on the SMP and the NUMA machines respectively. The Y-axis represents the observed sample median speedups. The X-axis of each figure represents the tested matrix sizes. For

each matrix size, we report the performance of the tested distinct thread pinnings. We consider five thread pinning strategies: `no affinity` (the baseline), `icc compact`, `icc scatter`, `LPGP/GPLP(RWM)` (does consider the read/write model) and `LPGP/GPLP(SM)` (does not consider the read/write model) strategies[4].

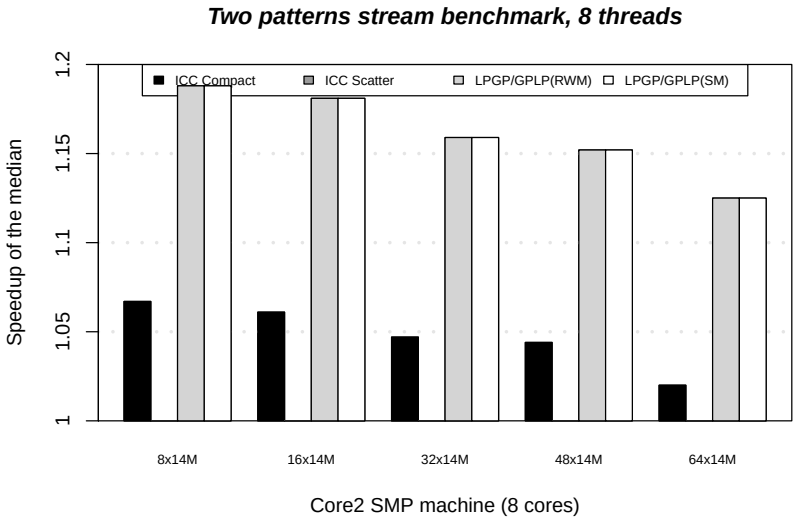


Figure 6.9: Speedup of the median of the tested thread affinities for the synthetic benchmark using multiple matrix sizes and running with 8 threads on `Intel` SMP machine. The baseline thread placement strategy is `no affinity`. Only statistically significant speedups are reported.

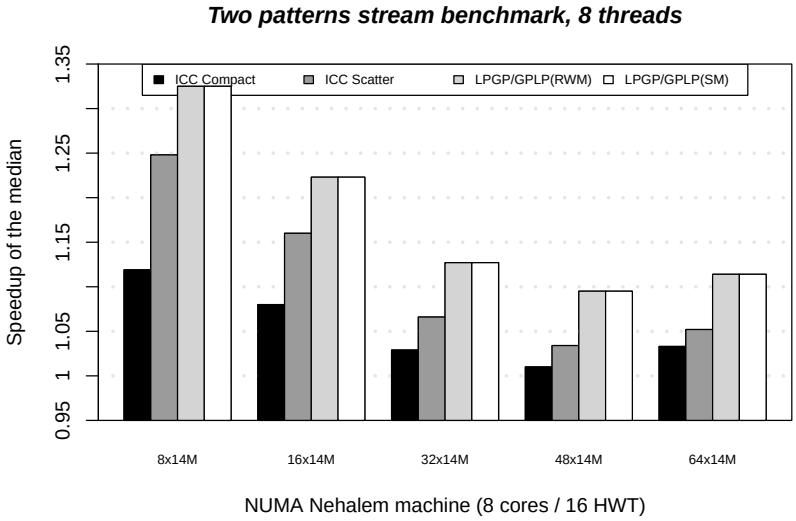


Figure 6.10: Speedup of the median of the tested thread affinities for the synthetic benchmark using multiple matrix sizes and running with 8 threads on `Intel` NUMA machine. The baseline thread placement strategy is `no affinity`. Only statistically significant speedups are reported.

First, it is clear that dynamic thread pinning is an effective technique to capture the inter-

[4]Even if the computed thread affinity using the RWM and the SM hierarchical strategies are similar, we choose to plot these strategies in separate bars.

thread data sharing exhibited by the two parallel regions. In fact, allowing thread migrations leads to the best program performance compared to application-wide thread pinning strategies.

Second, we observe that the speedups on the NUMA machine are larger than those on the SMP machine. Finally, we observe that while the `icc scatter` strategy achieves non-negligible observable speedups on the NUMA machine, it is not able to achieve similar performance improvement on the SMP machine. The explanation is simple: on the SMP machine, due to the presence of four separate L2 caches, the effect of applying `icc scatter` is to place threads on cores in a way that can not lead to exploit the sharing in the application whether in the first parallel region or in the second. Contrary to `icc scatter`, the `icc compact` strategy is able place threads on cores which lead to exploit some sharing but only on the second parallel region. Consequently, we observe speedups for this strategy on the SMP machine. On the NUMA machine, since it consists of two sockets where each has a larger L3 cache shared between four cores, applying the `icc scatter` strategy does exploit the sharing exhibited by the first parallel region, thus the statistical observed speedups.

Case study 2: each pair of threads share less than 100% of their assigned data
We previously showed how our synthetic benchmark behaves under some thread pinning strategies in function of matrix size. The benchmark is designed so that each thread accesses to the same amount of data. Moreover, the amount of shared data is equal between each pair of threads, of course with different sharing patterns across the two parallel regions. In order to analyse how the amount of inter-thread data sharing can influence the effectiveness of thread migrations across parallel regions, let us fix the same inter-thread data sharing in the first parallel region, and vary the amount of data sharing in the second. We consider in these experiments the 0%, 25% and 75% amounts of data sharing cases in the second parallel region.

Figures 6.11 and 6.12 show the obtained statistical median sample speedups on both the `Intel` SMP and NUMA machines respectively. We consider the two application-wide `icc` compiler strategies (white and gray bars respectively) and the per parallel regions affinity strategy (black bars). The baseline configuration is `icc compact`. These figures are organised as follows. Speedups are reported for each tested matrix size. Speedups are further reported according to the amount of data sharing in the second parallel region. Regardless of the amount of data sharing, we report speedups using three affinity strategies. This means that for each tested matrix size, we have two dimensions of plotting. The first dimension represents the tested distinct amounts of data sharing (100%, 75%, 25% and 0%). In the second dimension, we fix the amount of data sharing in the second parallel region and we report speedups for the three (`icc compact`, `icc scatter` and the per parallel regions thread affinity strategy) tested thread affinities. If we consider the case of the `icc compact` strategy (white bars), for each tested matrix size and reading from left to right, the first white bar represents the case of 100% data sharing, the second white bar represents the case of 75%, the third white bar represents the case of 25% and finally, the fourth white bar represents the case of 0% data sharing.

On the SMP machine, we observe that regardless of the amount of data sharing in the second parallel region, allowing the per parallel regions thread affinity leads to the best program performance. We can also observe that when there is no data sharing in the second parallel region the `icc scatter` obtains better performance than `icc compact`. With this sharing configuration, the two strategies achieve similar performance as far as the second parallel region is concerned (like thread migration strategy). However, when it comes to the first parallel region, `icc scatter` achieves better performance. Consequently, we obtain the observed program performance with `icc scatter`. On the NUMA machine, we observe that as the amount of

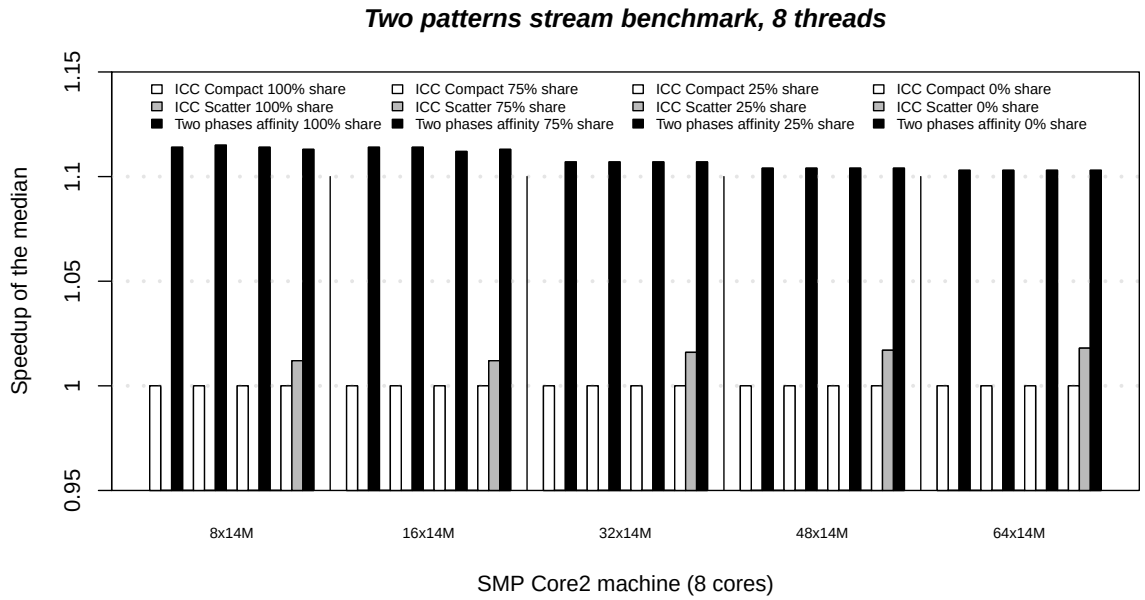


Figure 6.11: Speedup of the median of the tested thread affinities for the synthetic benchmark using multiple matrix sizes and running with 8 threads on `Intel` SMP machine. The baseline thread placement strategy is `icc compact`. White bars represent `icc compact`, gray bars represent `icc scatter` and black bars represent the per parallel regions thread affinity. For each tested matrix size, speedups are organised according to the amount of data sharing in the second parallel region (four groups of separated configurations) and the tested thread affinities (three strategies). Reading from left to right, the first group represents the case of 100% data sharing, the second group represents the case of 75% data sharing, the third group represents the case of 25% data sharing and the fourth group represents the case of 0% data sharing. Only statistically significant speedups are reported.

data sharing in the second parallel region is reduced, the performance of the `icc scatter` and per parallel regions thread affinity strategies are close. This is due to two reasons: 1) since these two strategies are able to exploit the data sharing of the first parallel region, performance for that parallel regions are similar, and 2) when there is no sharing at the second parallel region, the precise thread pinning is not important.

6.5.1.2 A matrix multiply benchmark

In this section, we present results for large and small size matrices.

Obtained results for large size matrices

Figures 6.13 and 6.14 report the observed program performance of the tested thread affinities of the matrix multiply benchmark running with 8 threads[5] on the `Intel` SMP (`Core2`) and NUMA (`Nehalem`) machines using bar plots. While the Y-axis represents the speedup of the sample median execution times, the X-axis of each figure represents the tested matrix sizes. For each matrix size, we report the speedup of a distinct thread pinning. For the per parallel regions computed thread affinity, we distinguish between thread pinnings computed using whether an

[5]We also tested the case of 16 threads on the `Intel` NUMA machine by enabling `Hyper-Threading` and conclude with the same observations as in the 8 threads experiments.

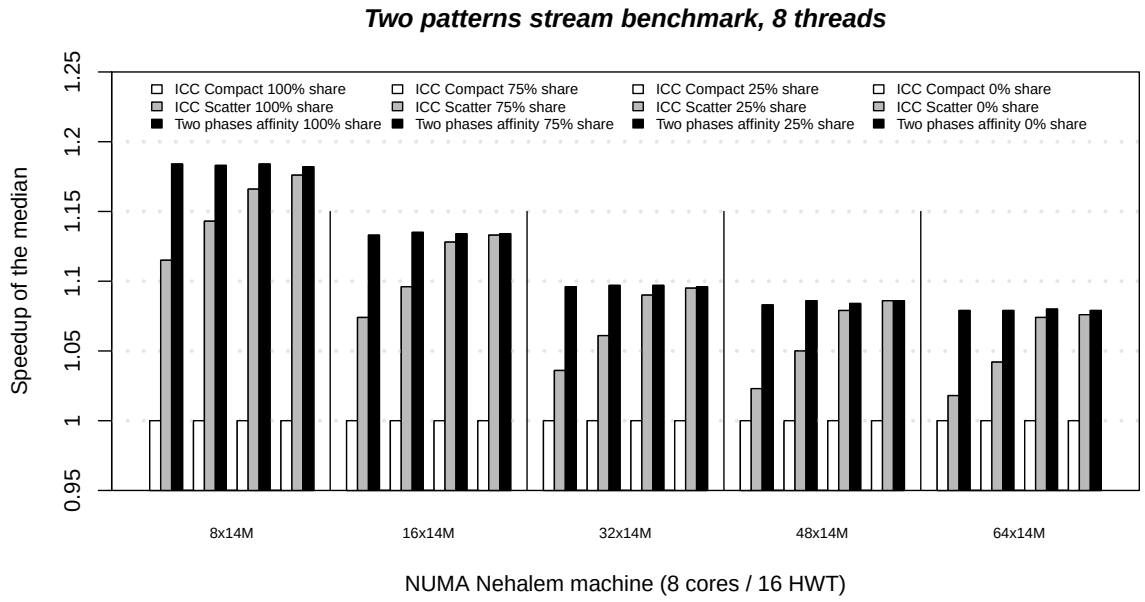


Figure 6.12: Speedup of the median of the tested thread affinities for the synthetic benchmark using multiple matrix sizes and running with 8 threads on `Intel` NUMA machine. The baseline thread placement strategy is `icc compact`. White bars represent `icc compact`, gray bars represent `icc scatter` and black bars represent the per parallel regions thread affinity. For each tested matrix size, speedups are organised according to the amount of data sharing in the second parallel region (four groups of separated configurations) and the tested thread affinities (three strategies). Reading from left to right, the first group represents the case of 100% data sharing, the second group represents the case of 75% data sharing, the third group represents the case of 25% data sharing and the fourth group represents the case of 0% data sharing. Only statistically significant speedups are reported.

aware or an unaware read/write model[6]. Moreover, we have to notice that the `LPGP(RWM)` and `GPLP(RWM)` strategies are the same. Similarly, we notice that the `LPGP(SM)` and `GPLP(SM)` strategies are equivalent. We conclude with the following observations:

1. On both machines, the performance of the `LPGP/GPLP(RWM)` strategies is similar to the performance of the `icc compact` strategy. Since B is a read only matrix, the read/write model is unable to capture the true sharing behaviour between threads. Consequently, the computed thread affinity of each parallel region is the same as the one set by the `Intel` compiler.

2. On both machines, the `LPGP/GPLP(SM)` strategy achieves the best program performance whatever the tested matrix size, this means that changing the affinity between parallel regions does improve program performance. The obtained speedups are in the range $[1.2, 1.7]$

3. On the `Intel` NUMA machine, we observe important speedups for all the tested thread affinities regardless of the tested matrix size. This situation can be simply explained by

[6]See Section 6.3.3 in page 120 for an aware read/write model and Section 5.2.2.2 in page 89 for an unaware read/write model.

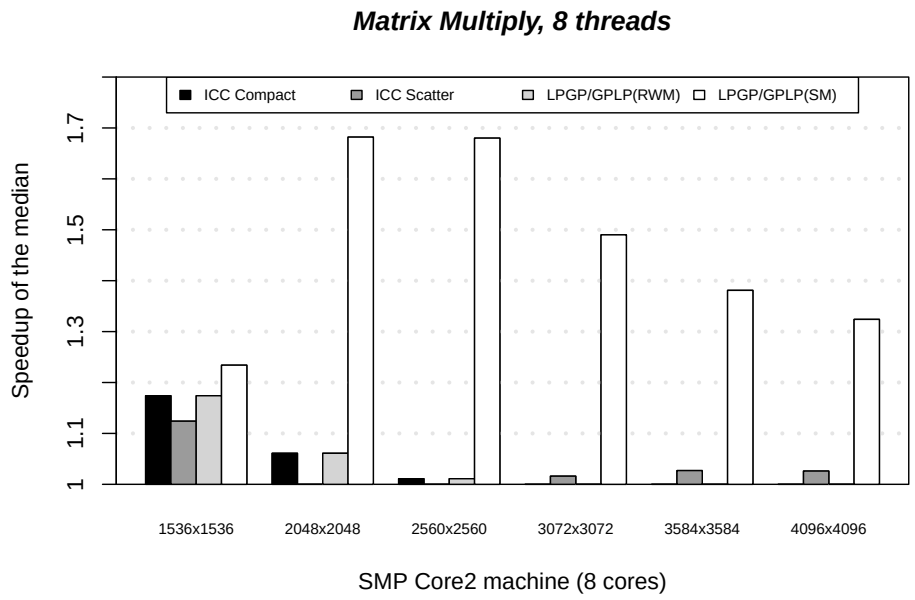


Figure 6.13: Speedup of the median of the tested thread affinities for the matrix multiply benchmark using multiple matrix sizes and running with 8 threads on `Intel` SMP machine. The baseline thread placement strategy is `no affinity`. Only statistically significant speedups are reported.

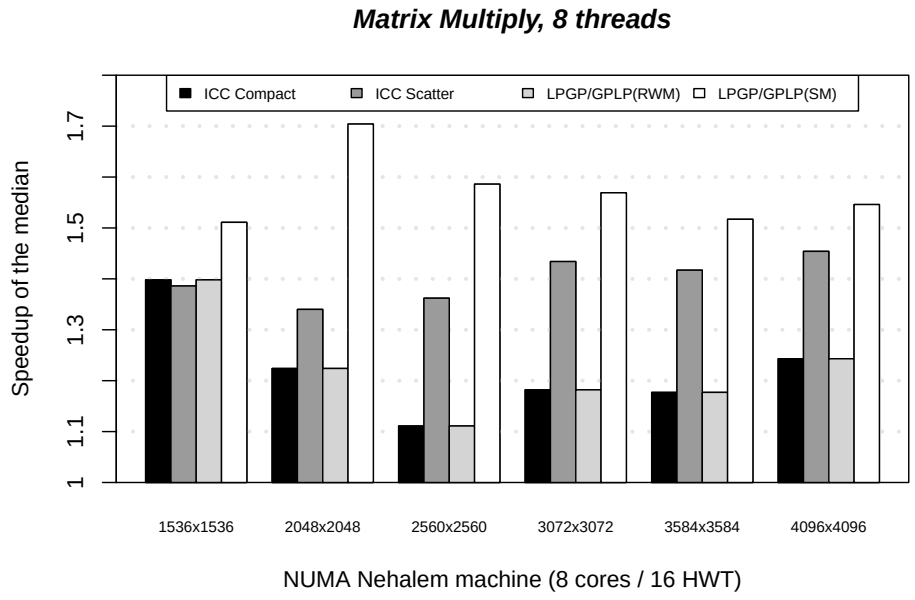


Figure 6.14: Speedup of the median of the tested thread affinities for the matrix multiply benchmark using multiple matrix sizes and running with 8 threads on `Intel` NUMA machine. The baseline thread placement strategy is `no affinity`. Only statistically significant speedups are reported.

the fact that the Linux scheduler does not behave very well in NUMA machines (it allows frequent thread migrations).

4. On the `Intel` SMP machine, except for the 1536×1536 matrix size, for all the tested matrix sizes, `icc compact` and `icc scatter` achieve either non statistically significant or negligible statistically significant speedups. This means that in terms of the sample median execution times, the Linux scheduler obtains good performance compared to `icc compact` and `icc scatter` for this benchmark.

Obtained performance for small size matrices

In the previous paragraph, we considered performance results for large matrix sizes. In other words, we considered only the case where the total working set of the benchmark does not fit in the available last level caches on both machines. Let us now present performance results for matrix sizes less than 1536. In this context, we vary the matrix sizes from 256 to 1536 (1.5MB to 54MB). For more accuracy of our measurements, we repeat the execution of each parallel region 1, 10, 50 and 100 times[7]. This means that we have to execute the whole first parallel region with all the iterations before executing the second parallel region with all its iterations as well as showed in Listing 6.5. By doing so, we want to check the relation between the computation granularity and the size of the working set. For large matrix sizes, it is obvious that the granularity of each matrix multiply is sufficient to change thread pinning. However, it is not clear if this situation is true for smaller size matrix sizes.

Listing 6.5: Program code for small matrices

```
for(i = 1; i <= NumberIterations; i++)
{
        ParallelMarixMultiply_1();
}
for(i = 1; i <= NumberIterations; i++)
{
        ParallelMarixMultiply_2();
}
```

Figures 6.15 and 6.16 report the observed speedups of the median execution times for the tested thread affinities regarding multiple small matrix sizes on both the `Intel` SMP and the NUMA machines respectively. We consider the two application-wide `icc` compiler strategies (white and gray bars respectively) and the per parallel regions affinity strategy (black bars). The baseline comparison is the `icc compact` strategy[8]. These figures are organised as follows. Speedups are reported for each tested matrix size. Speedups are further reported according to the number of repetitions of each parallel region. Regardless of the number of repetitions, we report speedups using three affinity strategies. This means that for each tested matrix size, we have two dimensions of plotting. The first dimension represents cases where the execution of each parallel region is repeated 1, 10, 50 and 100 times. In the second dimension, we fix the number of repetitions and we report speedups for the three (`icc compact`, `icc scatter` and the per parallel regions thread affinity strategy) tested thread affinities. If we consider the case

[7]Since we use small matrix sizes, the execution times are too short. Consequently, the benchmark may be sensitive to performance variability

[8]We have to notice that if we consider the non affinity configuration as a baseline, all the tested pinning strategies obtain important performance improvement

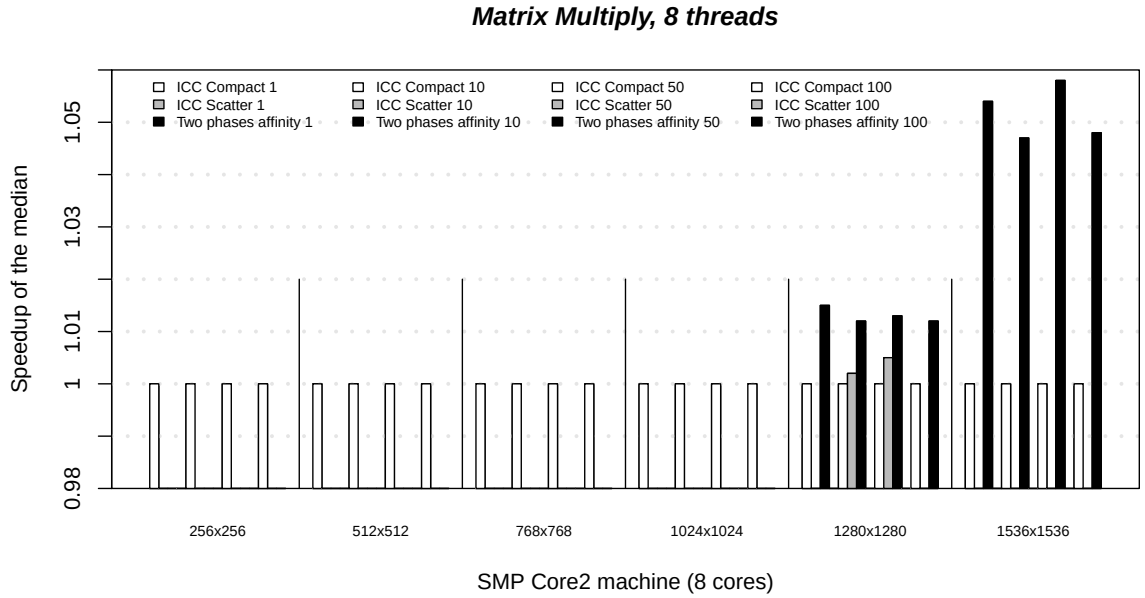


Figure 6.15: Speedup of the median of the tested thread affinities for the matrix multiply benchmark using multiple matrix sizes and running with 8 threads on `Intel` SMP machine. The baseline thread placement strategy is `icc compact`. White bars represent `icc compact`, gray bars represent `icc scatter` and black bars represent the per parallel regions thread affinity. For each tested matrix size, speedups are organised according to the number of repetitions of each parallel region (four groups of separated configurations) and the tested thread affinities (three strategies). Reading from left to right, the first group represents the case where each parallel region is repeated 1 time, the second group represents the case where each parallel region is repeated 10 times, the third group represents the case where each parallel region is repeated 50 times and the fourth group represents the case where each parallel region is repeated 100 times. Only statistically significant speedups are reported.

of the `icc compact` strategy (white bars), for each tested matrix size and reading from left to right, the first white bar represents the case where each parallel region is repeated 1 time, the second white bar represents the case where each parallel region is repeated 10 times, the third white bar represents the case where each parallel region is repeated 50 times and the fourth white bar represents the case where each parallel region is repeated 100 times. We conclude with the following observations:

1. On the `SMP` and `Nehalem` machines, we observe that for matrix sizes below 1280, we do not observe any performance benefit from applying the `icc scatter` or the per parallel regions thread affinity compared to the `icc compact` strategy. This means that neither the `icc scatter` nor the `LPGP/GPLP(SM)` strategies are adequate for small size matrices. Actually, this observation does not suggest that `icc compact` is better, it just says that we do not observe statistically significant performance difference.

2. We observe non negligible performance improvement for the per parallel regions thread affinity strategy for the 1280 and 1536 matrix sizes regardless of the number of iterations.

3. The per parallel regions thread affinity strategy is not effective for matrix sizes smaller than 1280 because the working set accessed by each thread fits completely on the last

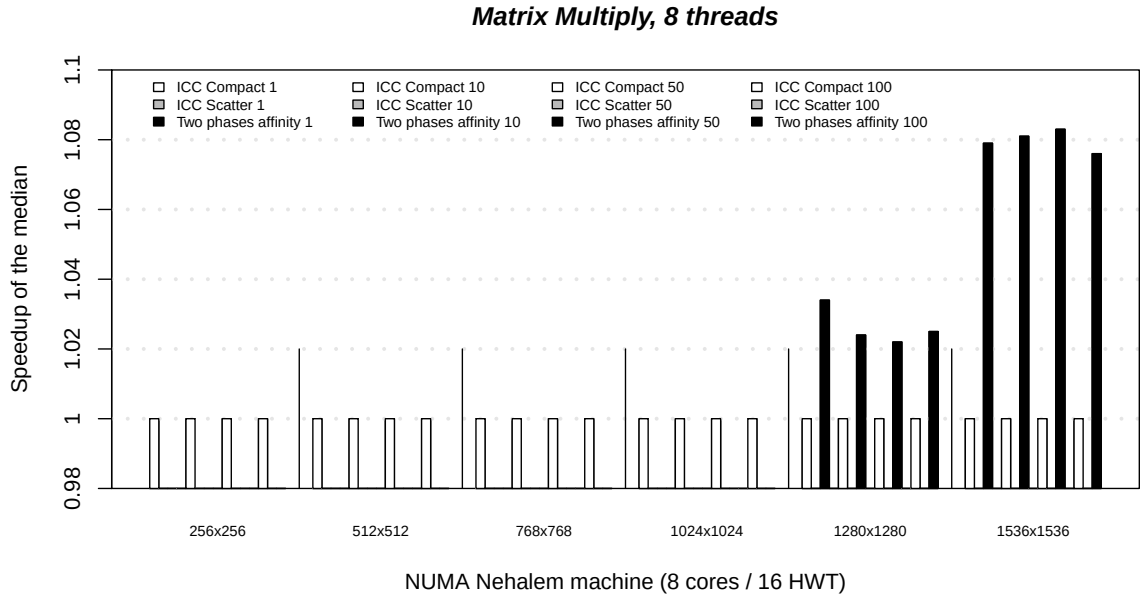


Figure 6.16: Speedup of the median of the tested thread affinities for the matrix multiply benchmark using multiple matrix sizes and running with 8 threads on `Intel` NUMA machine. The baseline thread placement strategy is `icc compact`. White bars represent `icc compact`, gray bars represent `icc scatter` and black bars represent the per parallel regions thread affinity. For each tested matrix size, speedups are organised according to the number of repetitions of each parallel region (four groups of separated configurations) and the tested thread affinities (three strategies). Reading from left to right, the first group represents the case where each parallel region is repeated 1 time, the second group represents the case where each parallel region is repeated 10 times, the third group represents the case where each parallel region is repeated 50 times and the fourth group represents the case where each parallel region is repeated 100 times. Only statistically significant speedups are reported.

level caches (4MB L2 cache on SMP machine and 8MB L3 cache on the NUMA machine). Moreover, matrix B (non-zero blocks) can be hosted almost in all the last level caches without harming the overall performance. Consequently, allowing thread migrations is not useful.

4. When we consider, the 1280 and 1536 matrix sizes[9], matrix B can not be hosted entirely in all the last level caches without harming the overall program performance. Consequently, the precise thread affinity for each parallel region is important, it permits to share only useful parts of B, thus preventing unnecessary cache misses.

We said earlier that for small matrices, non zero blocks of matrix B can be hosted in all the available last level caches. Let us analyse the situation of small matrices for the 1024×1024 matrix size case on the `Intel` SMP machine. This machine has four L2 caches of 4 MB each. When we decompose each matrix into 4×4 blocks, each block has a 0.5 MB size. With matrices decomposed into 16 blocks and 8 threads, each thread has to compute two blocks of matrix C. Since computing a block of C requires one block in A and another of B, work distribution in the benchmark assigns for each thread, two blocks of C, two blocks of A and 2 blocks of B.

[9]We have to notice that we tested intermediate size matrices, we conclude with similar observations.

Let us now measure the footprint for each L2 cache. With an L2 cache shared between each pair of cores, and with a number of cores equal to the number of threads, each L2 cache has to host the working set of two threads in order to compute two distinct blocks in C. With a 0.5 MB block size, each thread has to access to a 0.5 MB size block of C and a 0.5 MB size block of A, what makes 1 MB. The footprint for two threads is 2 MB. Matrix B has four non zero blocks, which makes 2 MB of memory cache footprint. Consequently, at most there are 4 MB (2 MB from A and C and at most 2 MB from B) of working set in each L2 cache (an L2 cache has 4 MB), sufficiently enough to compute two distinct blocks in C. Because all L2 caches can host the non zero blocks of B, the exact pinning is not important to take any benefit from sharing some parts of matrix B. That is why we do not observe any performance benefit from allowing thread migration between the two parallel regions.

Obtained results for an iterative behaviour for the matrix multiply benchmark
In our benchmark, we considered only the case where, regarding the parallel regions control flow, the second matrix multiply (2^{nd} phase) is executed after the first one before the benchmark resumes its execution. That is, the expected performance behaviour when we execute the two matrix multiplications in a iterative way is unclear. In order to answer this question, we performed a set of experiments in which, after fixing a matrix size and a thread affinity strategy, we repeat in an iterative way the execution of the two matrix multiplications. In these experiments, the considered number of iterations is 1, 2, 4, 6, 8 and 10 (see Listing 6.6).

Listing 6.6: Iterative behaviour of two phases matrix multiply

```
for(i = 1; i <= NumberIterations; i++)
{
        ParallelMarixMultiply_1();
        ParallelMarixMultiply_2();
}
```

Figures 6.17 and 6.18 report the observed speedups of the median execution times for the tested thread affinities regarding some tested matrix sizes on both the `Intel` SMP and the NUMA machines respectively. We consider the two application-wide `icc` compiler strategies (white and gray bars respectively) and the per parallel regions affinity strategy (black bars). The baseline comparison is the `icc compact` strategy[10]. These figures are organised as follows. Speedups are reported for each tested matrix size. Speedups are further reported according to the number of iterations of the two parallel regions. Regardless of the number of iterations, we report speedups using three affinity strategies. Again, this means that for each tested matrix size, we have two dimensions of plotting. The first dimension represents cases where the number of iterations (for each iteration, the two parallel regions are executed) in the program is 1, 2, 4, 6, 8 and 10. In the second dimension, we fix the number of iterations and we report speedups for the three (`icc compact`, `icc scatter` and the per parallel regions thread affinity strategy) tested thread affinities. Using the case of the `icc compact` strategy (white bars) for illustration purpose, for each tested matrix size and reading from left to right, the first white bar represents the case where the number of iterations is equal to 1, the second white bar represents the case where the number of iterations is equal to 2, until the sixth white bar which represents the case

[10]We have to notice, that if we consider the non affinity configuration as a baseline, the obtained speedups are far more significant

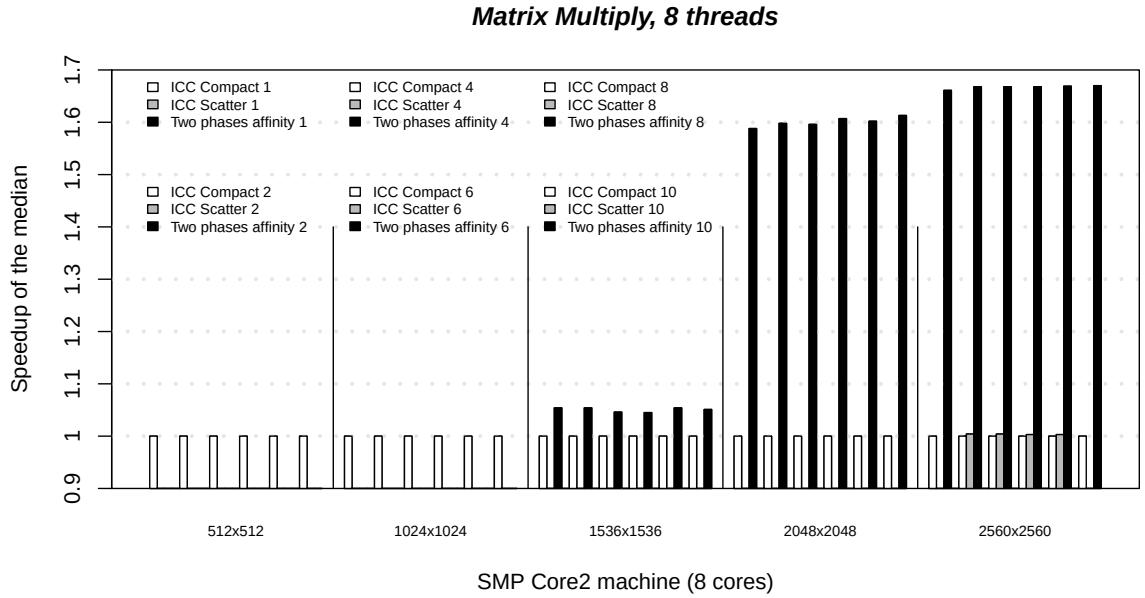


Figure 6.17: Speedup of the median of the tested thread affinities for the matrix multiply benchmark using multiple matrix sizes and running with 8 threads on `Intel` SMP machine. The baseline thread placement strategy is `icc compact`. White bars represent `icc compact`, gray bars represent `icc scatter` and black bars represent the per parallel regions thread affinity. For each tested matrix size, speedups are organised according to the number of executions (iteratively) of the two parallel regions (six groups of separated configurations) and the tested thread affinities (three strategies). Reading from left to right, the first group represents the case where the execution of the two parallel regions is repeated 1 time, the second group represents the case where the execution of the two parallel regions is repeated 2 times, the third group represents the case where the execution of the two parallel regions is repeated 4 times, the fourth group represents the case where the execution of the two parallel regions is repeated 6 times, the fifth group represents the case where the execution of the two parallel regions is repeated 8 times and the sixth group represents the case where the execution of the two parallel regions is repeated 10 times. Only statistically significant speedups are reported.

where the number of iterations is equal to 10.

First, we can observe that on both machines, the observed speedups are not statistically significant for matrix sizes below than 1536. Second, when matrix size is equal or larger to 1536, the observed speedups are not negligible. Finally, we observe that whatever the tested number of iterations[11], the obtained speedups with larger matrices are almost the same. This means that iteratively migrating threads through the two parallel regions does not degrade the overall program performance. Therefore, this strategy provides better program performance than application-wide strategies as long as the amount of work in each parallel region is important to overcome the overhead inherent to thread migrations.

The next section presents obtained performance results using SPEC OMP01 and NPB benchmarks.

[11] We also tested a number of iterations equal to 16, 32, and 64, the conclusions are similar

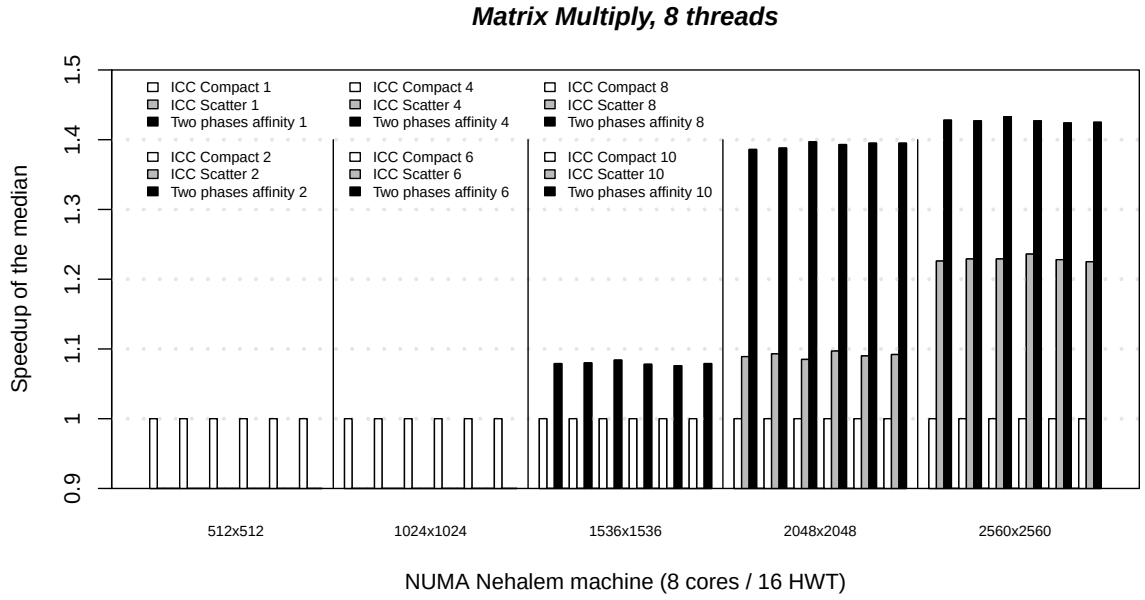


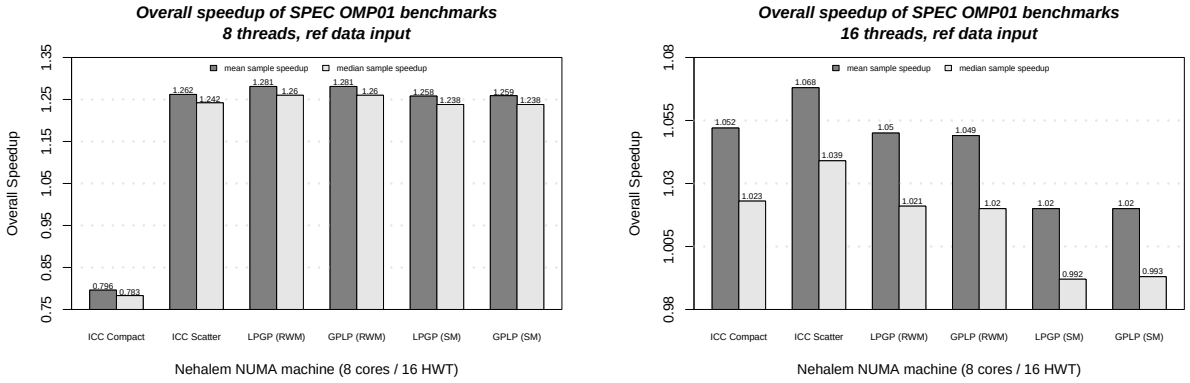
Figure 6.18: Speedup of the median of the tested thread affinities for the matrix multiply benchmark using multiple matrix sizes and running with 8 threads on `Intel` NUMA machine. The baseline thread placement strategy is `icc compact`. White bars represent `icc compact`, gray bars represent `icc scatter` and black bars represent the per parallel regions thread affinity. For each tested matrix size, speedups are organised according to the number of executions (iteratively) of the two parallel regions (six groups of separated configurations) and the tested thread affinities (three strategies). Reading from left to right, the first group represents the case where the execution of the two parallel regions is repeated 1 time, the second group represents the case where the execution of the two parallel regions is repeated 2 times, the third group represents the case where the execution of the two parallel regions is repeated 4 times, the fourth group represents the case where the execution of the two parallel regions is repeated 6 times, the fifth group represents the case where the execution of the two parallel regions is repeated 8 times and the sixth group represents the case where the execution of the two parallel regions is repeated 10 times. Only statistically significant speedups are reported.

6.5.2 Performance analysis using SPEC OMP01 and NPB benchmarks

In this section, we present the results of our performance evaluation regarding the effectiveness of dynamic thread pinnings for SPEC OMP2001 and NPB applications on NUMA machines. We run each benchmark multiple times under multiple thread pinning strategies. For the purpose of the evaluation, we used three NUMA machines: the `Nehalem` machine with 8 (16 hardware threads) cores, the `Shanghai` machine with 8 cores and the `Barcelona` machine with 16 cores. Each benchmark was run with 8 and 16 threads with respect to the maximal number of physical cores. We also tested the case of the 16 threads on the `Nehalem` machine by using all the hardware threads[12]. Besides, we used the `ref` and `Class B` data inputs for SPEC OMP and NPB respectively, for both memory trace collection and the performance measurement when applying different thread pinnings.

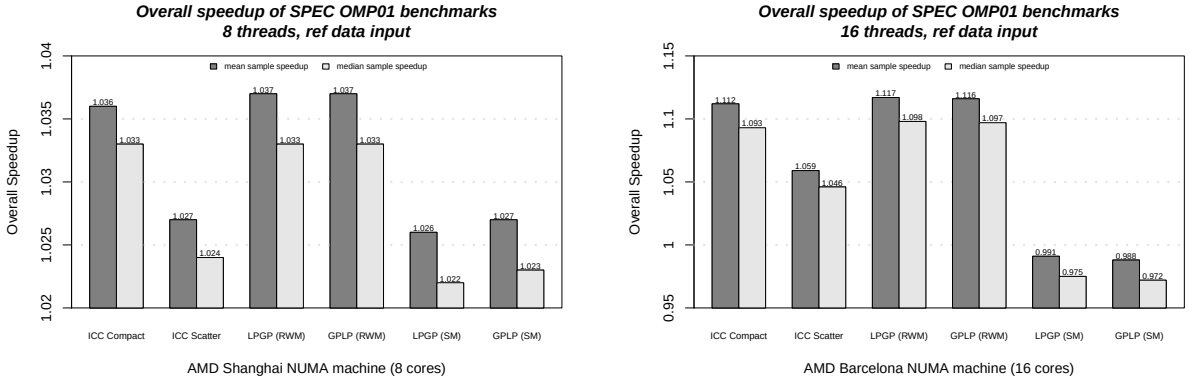
[12]Hyper-Threading enabled

6.5.2.1 Experimental results



(a) (b)

Figure 6.19: Overall sample speedups of the tested thread affinities with SPEC OMP2001 benchmarks running on the `Intel` NUMA machine. The baseline thread placement strategy is the OS free affinity.

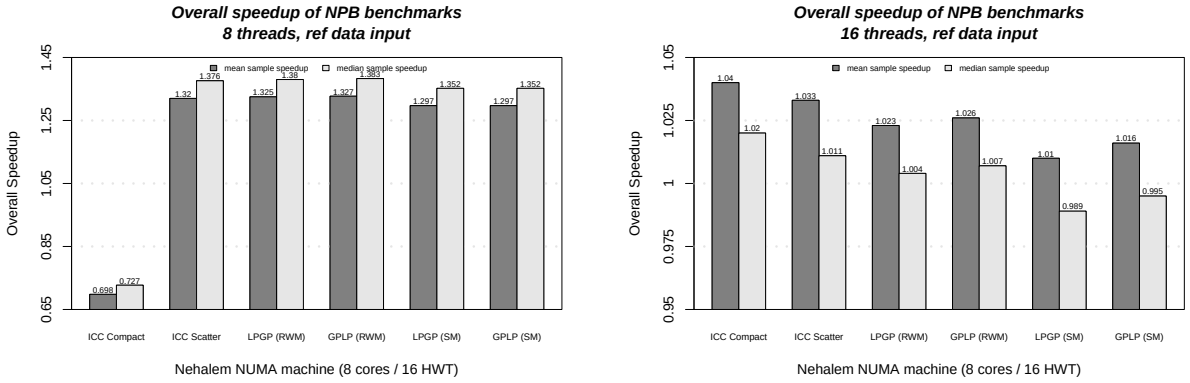


(a) (b)

Figure 6.20: Overall sample speedups of the tested thread affinities with SPEC OMP2001 benchmarks running on the `AMD` NUMA machines. The baseline thread placement strategy is the OS free affinity.

Three figures reflect the speedups through the tested thread pinning strategies (`icc compact`, `icc scatter`, `LPGP(RWM)`, `GPLP(RWM)`, `LPGP(SM)` and `GPLP(SM)`) compared to the default `no affinity` strategy of the OS scheduler. Figure 6.19 and 6.20 show the overall sample speedups of every thread pinning strategy on the `Nehalem` and the `AMD` NUMA machines using bar plots. We report the speedups of the average and the median execution times of all SPEC OMP applications running with 8 and 16 threads. Similarly, Figure 6.21 illustrates the same performance metrics on the `Nehalem` machine for NPB benchmarks. For each figure, while each bar in the X-axis represents a distinct thread pinning strategy, the Y-axis reports the observed sample speedups. We conclude the following:

1. On the `Nehalem` and `Barcelona` machines, running OMP01 and NPB with 16 threads with thread affinity enabled, leads to marginal speedups and slowdowns (Figures 6.19b, 6.20b and 6.21b).



(a) (b)

Figure 6.21: Overall sample speedups of the tested thread affinities with NPB benchmarks running on the `Intel` NUMA machine. The baseline thread placement strategy is the OS free affinity.

2. On the `Shanghai` machine, fixing thread affinity for OMP01 running with 8 threads leads to marginal speedups (Figure 6.20a).

3. On the `Nehalem` machine, when running OMP01 and NPB with 8 threads (Figures 6.19a and 6.21a), except the `icc compact` strategy, we observe non-negligible speedups for all the tested strategies. The reason is that the experiments were performed with `Hyper-Threading` (HT) enabled. Enabling this option increases the number of possible OS scheduling possibilities and since the OS scheduler gives higher priority for load balancing, the direct result is an increase in thread migrations and consequently a poor overall performance is observed[13]. This observation explains the poor performance of the `icc compact` strategy as well. Indeed, when HT is enabled, the `icc compact` strategy places all the 8 threads on a single socket[14].

4. Even if the difference in terms of speedups is not significant, we observe that the `LPGP(SM)` and the `GPLP(SM)` produce poor performance compared to `LPGP(RWM)` and `GPLP(RWM)` strategies. As a reminder, while the former strategies (SM) are computed from affinity graphs that do not consider the read/write model, the later strategies (RWM) are computed from affinity graphs that do consider a read/write model[15].

5. Regarding the test machines, we do not observe any important difference, in terms of speedups, between the tested thread affinity strategies. This situation may suggest that there is no benefit in enabling a per parallel regions thread affinity. Moreover, it is possible to conclude that this approach is not effective for SPEC OMP01 and NPB benchmarks.

[13]We also performed other experiments, thanks to `cpusets`: we constrain threads to run only on a subset of cores (as if there are 8 cores on the machine without HT). In this configuration, the observed speedups were marginal.

[14]If we consider the case the `icc compact` strategy which distributes threads evenly on both sockets (4 threads per socket) then, the observed performance with this strategy is very close to `LPGP(RWM)`

[15]See Section 6.3.3 in page 120 for an aware read/write model and Section 5.2.2.2 in page 89 for an unaware read/write model.

6.5.2.2 Discussion

As noticed in the previous section, from our experiments, we made two main observations which are highly related:

1. The relative poor performance of strategies computed upon a model which does not consider the read/write model. If we consider strategies that do not consider the RWM, the observed overall sample speedup of the median is the range $[0.972 - 1.352]$. On the other hand, strategies that do consider the RWM have an overall sample speedup of the median in the range $[1.004 - 1.383]$.

2. The non clear benefit of enabling a per parallel regions thread affinity. If we compare the best overall sample speedup of the median obtained by application independent and application dependent strategies for each tested configuration (tested machine, number of threads and benchmarks suit), we observed that while application independent strategies have speedups in the range $[1.033 - 1.376]$, application dependent strategies have speedups in the range $[1.007 - 1.26]$. Even if the differences between the obtained speedups are not important, it is possible to conclude that per parallel regions affinity is not effective compared to application-wide (application independent) strategies.

In order to understand the presented experimental results, we first show in Table 6.1 the total number of times for which the computed per parallel regions thread affinity (the `LPGP` and `GPLP` strategies computed using whether an aware or an unaware read/write model) consists of an application-wide thread affinity. This means, that the computed thread affinity is almost the same for all the parallel regions (Tables 6.2 and 6.3 report the number of parallel regions in all the studied benchmarks), or at least for the detected most time consuming parallel regions. From Table 6.1, we can observe that using `LPGP(RWM)` and `GPLP(RWM)` strategies, at least half of the benchmarks (either OMP01 or NPB) were run with a single (application-wide) thread affinity. On the other hand, the `LPGP(SM)` and `GPLP(SM)` strategies do not seem to reflect the same behaviour. Indeed, for the SM strategies, we can observe that almost all the computed per parallel regions thread affinity have a thread affinity computed for at least two parallel regions.

In the light of the previous observations, we can say as a first conclusion, that thread affinity strategies computed from affinity graphs that do consider read/writes, better capture the sharing behaviour of threads than strategies that do not consider read/writes. Consequently, the thread pinnings computed with these strategies are more likely to lower the number of possible thread migrations. Moreover, since almost all the per parallel regions thread affinity computed with a read/write model tend to be application-wide strategies, explains why we observe that the performance of the RWM strategies is close to the performance of strategies like `icc compact` or `icc scatter`[16].

Now, we have to understand why strategies that exhibit distinct thread pinnings for distinct parallel regions are less effective compared to application-wide strategies. There are mainly two reasons for this performance behaviour given in a decreasing order of importance:

1. The ratio between the number of times each parallel region is called, and the execution time elapsed in the execution of a single iteration of a given parallel region is very low (as noticed before in Tables 6.2 and 6.3). Consequently, the inherent overhead from frequent thread migrations and due to the small granularity of the selected *hot* parallel regions lead to lower the benefit from that migrations.

[16] We observed in Chapter 5 that there is no any important difference, in terms of speedups, between application independent and application dependent strategies.

Benchmarks suit	#Threads	Machine	LPGP (DM)	GPLP (DM)	LPGP (SM)	GPLP (SM)
SPEC OMP01	8	Nehalem	10/10	10/10	3/10	3/10
	8	Shanghai	10/10	10/10	3/10	3/10
	16	Nehalem	5/10	5/10	4/10	4/10
	16	Barcelona	10/10	10/10	4/10	5/10
NPB	8	Nehalem	5/8	7/8	4/8	4/8
	16	Nehalem	3/8	3/8	3/8	3/8

Table 6.1: Number of benchmarks where the computed per-parallel region thread affinity consists of setting a single-global-wide thread affinity. Each benchmark is executed using 8 and 16 threads on the `Nehalem` and/or the `Shanghai` and `Barcelona` machines

2. The poor inter-thread data sharing exhibited by the distinct parallel regions for the tested benchmarks. Thus, applying the dynamic thread affinity technique on SPEC OMP01 and NPB benchmarks leads to the observed poor program performance. Unfortunately, this is true (see Chapter 5) as a direct consequence of: 1) the uniform distribution of the working set between threads and 2) the presence of non-uniform data sharing patterns is rare.

Benchmarks	#Parallel regions	#Iterations
wupwise	10	402
swim	8	1198
mgrid	12	18250
applu	22	50
galgel	32	117
equake	11	3334
apsi	24	50
fma3d	30	522
art	4	1
ammp	10	202

Table 6.2: Number of parallel regions in SPEC OMP01 benchmarks running with the `ref` data input. For each benchmark, the number of iterations of the first hot parallel region is reported.

Benchmarks	#Parallel regions	#Iterations
bt	9	202
cg	7	75
ep	3	1
ft	8	22
lu	9	251
mg	10	170
sp	13	402
ua	55	2251

Table 6.3: Number of parallel regions in NPB benchmarks running with the `CLASS B` data input. For each benchmark, the number of iterations of the first hot parallel region is reported.

6.6 Conclusion

We have presented an approach to exploit phase-based behaviour in OpenMP programs using thread affinity. The presented technique rely on the *control flow graph* of the parallel OpenMP regions. The *control flow graph* gives for each parallel region its predecessor and successor in the execution flow. In other words, it is the graph representing the execution flow of distinct parallel regions. We have extended an existing tool to instrument the OpenMP constructs. Using a binary instrumentation tool, we build an *affinity graph* for each parallel region in the program. After that, we compute multiple thread pinning strategies for each parallel region.

Besides the computed thread affinity for each parallel region (from its *affinity graph*) in the program, the extracted *control flow graph* of the parallel regions gives also information about

the call frequency of each parallel region. We think that the later information is valuable when it comes to exploit dynamic thread pinning. Indeed, combined with a timing measurement of each parallel region, the call frequency allow us to focus only on hot parallel regions. Therefore, we prevent thread migration on short parallel regions which may degrade performance.

We investigate the effectiveness of a per-parallel regions thread affinity using various *affinity graphs* decomposition techniques for SPEC and NPB OpenMP programs. We also tested the performance of some synthetic micro-benchmarks which capture some non-frequent inter-thread sharing patterns. Using the application independent `no affinity`, `icc compact` and `icc scatter` strategies and the application dependent `LPGP` and `GPLP` strategies, we performed a statistical performance evaluation and analysis. We conclude, that as far as SPEC OMP01 and NPB benchmarks are concerned, they do not really take benefit from dynamic thread pinning compared to the obtained benefit by using only application-wide (either application dependent or application independent) strategies for example. This does not suggest that this approach degrades performance in a great extent. Actually, we observed marginal speedups and slowdowns for almost all the tested strategies in SPEC and NPB benchmarks. Consequently, using a statistical significance analysis, as far as thread affinity is used, the obtained program performance is better than the Linux free strategy.

Regarding the tested synthetic micro-benchmarks, the conclusions are different. Indeed, this class of applications effectively take benefit from the per-parallel region thread affinity approach. For the all tested micro-benchmarks the performance difference between application-wide strategies and the dynamic thread pinning approach can not be considered as negligible. This is due to the high amount of inter-thread data sharing and to the implemented distinct sharing patterns.

The factors that influence the most on the effectiveness of the dynamic thread pinning technique, are the call frequency information, the granularity of a distinct parallel region and the amount of data sharing. Too short execution times, combined with a high number of calls to each parallel region produces inevitably a high overhead due to frequent thread migrations even with a presence of an important inter-thread data reuse ratio in each parallel region. One way to overcome this limitation is to aggregate consecutive parallel regions. We mean the fusion of that regions (in terms of memory tracing and affinity computation) so that they can be considered as a unique parallel region of code. By doing so, we can compute a thread affinity for the merged parallel regions and consequently reduce the number of possible thread migrations.

Chapter 7

Conclusion

In this thesis, we studied the performance behaviour of multi-threaded OpenMP applications running on multicore processors in terms of program execution times stability and shared cache performance improvement. In the first part, we analysed the variability of program execution times of OpenMP applications under multiple software configurations. We followed a methodology in which after fixing the experimental setup, we stressed some micro-architectural, architectural and OS parameters. For performance analysis, we used a rigorous statistical performance evaluation approach with the aim to quantify and qualify factors that influence the most on performance stability.

In the second part, we studied the impact of inter-thread data sharing on the performance of multi-threaded applications. For this purpose, we used thread affinity as a technique to exploit that sharing. In fact, while we know the impact of thread affinity in terms of performance stability, we further studied its constructive or destructive impact in terms of program execution improvement. To do so, we rigorously studied the impact of exploiting the inter-thread data sharing exhibited by OpenMP applications using various thread affinity strategies on the overall program performance. In addition, depending on data characteristics of the parallel application, the benefit from application-wide thread pinning strategies taking into account affinity relationships between threads is obvious. However, it is possible to further exploit inter-thread data sharing using thread affinity at different levels in the program through a single run. For this reason, we also proposed a technique to exploit the phase behaviour exhibited by OpenMP applications in order to compute effective thread placements strategies and allowing migrations to improve the whole program execution time. In this conclusion we first summarise our findings and contributions before discussing future work and perspectives.

7.1 Contributions

Variability of program execution times is an important factor to consider when it comes to study the performance of programs. Indeed, underestimating this problem can lead to misleading conclusions about the true performance behaviour of programs. Using a rigorous experimental methodology and a statistical performance evaluation protocol [TWB10], we performed an extensive experimental study that aims to quantify and qualify the factors that influence the most on variability of program execution times. Besides, we define a metric that measures this variability, it measures the disparity between extrema observations or outliers.

Our extensive experimental study demonstrate that, contrary to sequential long running applications, OpenMP applications exhibit a variability of program execution times up to 30% on

SMP multicore platforms. We showed that even if it is not unique, using thread affinity removes the performance variability in most of the cases if the number of threads does not exceed the number of cores. Yet, other factors such as automatic hardware prefetching, the size of memory pages or the OS time slice still play an important role on performance stability. These factors can contribute to increase performance variability of OpenMP programs from 1% to 40% in multicore platforms.

As reported by our performance variability study, thread affinity plays an important role in terms of performance stability. To check how thread affinity improves or degrades the overall program performance, we investigate the performance of various application-wide thread pinning strategies for SPEC and NPB OpenMP applications. In our study, we considered cache-unaware strategies [1] and cache-aware strategies [2]. For strategies that rely on the characteristics of the application, we use a profile guided method that produces an *affinity graph* representing the relationship between each pair of threads in terms of data sharing. Once an affinity graph constructed, we use graph partitioning techniques to compute effective thread affinity strategies as far as data sharing is concerned.

Our statistical performance evaluation and analysis demonstrated that fixing an affinity provides statistically significant speedups in the range of $[1 - 2.2]$ compared to the Linux OS strategy. However, while the performance improvement on UMA (Uniform Memory Access) machines is marginal, on NUMA (Non Uniform Memory Access) machines it is significant. We also showed that cache-unaware thread affinity strategies provide equivalent performance gains compared to cache-aware strategies. Of course, this observation does not suggest that we do not need precise thread affinity strategies. Still, we have to distinguish between three situations. First, cache-aware strategies provide better performance improvements in benchmarks with significant inter-thread data sharing. Second, most of the tested OpenMP applications do not exhibit enough data sharing to observe any important difference between cache-aware and cache-unaware thread pinning strategies. Finally, the study focuses only on cache effects. But there are other factors that may influence the performance obtained by precise affinity strategy. For instance, we can consider bus contention, prefetch contention, last level cache contention, memory controller contention or OS memory pages allocation.

In addition, we think that profile guided methods should be better if they consider program phases to decide about variable thread pinnings. To do so, we exploit the fact that most often, OpenMP programs implement multiple parallel regions with possibly distinct sharing patterns. Indeed, on smaller code fragments, the computed thread affinity can be more effectively translated into shared data. Consequently, applying this affinity may lead to better program performance. In order to compute a distinct thread affinity for each parallel region, we have extended the `OPARI` instrumentation tool to instrument the OpenMP parallel region constructs. Using binary instrumentation, we build an *affinity graph* for each parallel region in the program. After that, we compute multiple thread pinning strategies for each parallel region.

Using a statistical performance evaluation and analysis, we investigate the performance of the per-parallel regions thread affinity using SPEC and NPB OpenMP programs and some synthetic micro-benchmarks. Regarding SPEC OMP01 and NPB benchmarks, we have showed that there is no performance difference between the per-parallel regions approach and the application-wide approach. From a statistical significance analysis perspective, and as far SPEC and NPB appli-

[1] We apply the same strategy for all the tested benchmarks.
[2] We compute a distinct strategy for each benchmark based on its data characteristics.

cations are concerned, we observed marginal speedups and slowdowns for almost all the tested strategies. The reason is mainly due to the limited amount of inter-thread data sharing, and to the uniform distribution of the working set between threads. This is obvious since these standard benchmarks are written to target symmetrical multiprocessors. This means that multi-threaded applications have to consider these factors in their design when it comes to improve cache performance of multicore platforms using thread affinity. On the other hand, regarding the tested synthetic micro-benchmarks, the conclusions are different. Indeed, these applications effectively take benefit from the per-parallel regions thread affinity approach. For the all tested micro-benchmarks, the performance difference between application-wide strategies and the dynamic thread pinning approach are important. Of course, this is due to the high amount of inter-thread data sharing and to the implemented distinct sharing patterns. Moreover, we have performed a performance sensitivity analysis to demonstrate the conditions required for the per-parallel regions thread affinity technique to be effective. Our analysis shows that factors that influence the most are the call frequency information, the amount of data sharing and the granularity of a distinct parallel region. In fact, a parallel region that executes for a relatively too short execution time (compared to the overhead of migrating threads) combined with a high number of calls to each parallel region, leads to frequent thread migrations which inevitably degrade program performance.

7.2 Perspectives

As a natural extension of this work, we can consider two complementary aspects that we will detail later: 1) better application characterisation by the compiler, and 2) effective OS monitoring policies. Furthermore, accurate performance prediction models are necessary in order to tackle the challenges inherent to new many-core architectures.

The first aspect that we discuss is better application characterisation by the compiler. Through static code analysis, a compiler can achieve data sharing detection and quantification or code restructuring in order to account for the cache topology of multicore processors. With this application characterisation, extra code can be added to the generated code which may handle whether thread placement on cores or memory pages placement on memory nodes. Of course, this thread/data placement needs information about the topology of the machine which can be easily retrieved nowadays by many libraries like HWLOC [BCOM$^+$10]. Even if this approach requires more intention, it is actually not easy to perform an accurate data characterisation since, information like data input and number of threads are only known at runtime. Furthermore, compilers are not as effective with irregular codes as with regular data access patterns. To overcome this limitation, we think that it is possible to generate extra code, which at runtime does data sharing characterisation and thread/memory page placement. Another alternative, could be to define some API allowing the application layer to communicate the compiler about data sharing and data distribution.

The second aspect that we can consider is with effective OS monitoring policies. We discussed the advantages and drawbacks of a full compiler approach. We think that it can be augmented with an operating system support for overall system performance optimisation. In fact, as a resource manager, the operating system can play an important role in monitoring and satisfying the requirements in terms of shared resources of all the tasks running concurrently in the system by enforcing better and intelligent scheduling policies. To achieve this objective, we think that there are two complementary strategies. Indeed, as the number of competing tasks increases, the main problem is to identify the requirements and the affinities between them

at runtime and to enforce the adequate scheduling policy. One approach is to allow the OS monitoring all the running threads using hardware performance counters present in almost all modern multicore processors. These monitoring units can be used to approximate for instance the needs in terms of inter-thread cache sharing or inter-thread memory pages sharing. Due to the non-negligible overhead of this monitoring work, an alternative consists of extending the programing interface of the OS. To do so, it is possible to use some special system calls to express the resources requirements of each application/thread in terms of data sharing or any other optimisation metric. Once the OS identifies the requirements of all the running applications on the system, it is possible to dynamically apply scheduling policies that aims to improve the overall system performance.

In the light of the needs for more coordination between compilers and operating systems, a better performance understanding of future multicore architectures is necessary. Indeed, in addition to the increasing complexity of memory hierarchy, we witnessed the shift from symmetrical to asymmetrical designs. General purpose multicore processors are supported by a variety of accelerators like GPUs and FPGAs requiring special programming environments and highly intrusive code modifications of existing applications. Moreover, we assist now to the emergence of new accelerators designs like the `Intel` MIC architecture which can be programmed using OpenMP for instance. To effectively achieve sustainable performance, programming models like hybrid models, computing environments or operating system scheduling policies have not to consider anymore that computing units are homogeneous. In fact, they have to effectively be able to map the hierarchical parallelism exhibited by parallel applications into the hierarchical multicore machines.

Bibliography

[ATSS09] Reza Azimi, David K. Tam, Livio Soares, and Michael Stumm. Enhancing operating system support for multicore processors by using hardware performance monitoring. *SIGOPS Operating Systems Review*, 43(2):56–65, 2009.

[AW03] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, (HPCA '03), page 7, Washington, DC, USA, 2003. IEEE Computer Society.

[BCOM+10] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a generic framework for managing hardware affinities in HPC applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, pages 180–186, Pisa, Italia, February 2010. IEEE Computer Society Press.

[BD01] K. Beyls and E.H. D'Hollander. Reuse distance as a metric for cache behavior. In T. Gonzalez, editor, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 617–622, Anaheim, California, USA, 8 2001. IASTED.

[BDG+00] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, (Supercomputing '00), page 42, Washington, DC, USA, 2000. IEEE Computer Society.

[Bel97] Frank Bellosa. Follow-on scheduling: Using TLB information to reduce cache misses. In *Sixteenth Symposium on Operating Systems Principles (SOSP '97), Work in Progress Session*, Saint Malo, October 5–8 1997.

[BH04] E. Berg and E. Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, (ISPASS '04), pages 20–27, Washington, DC, USA, 2004. IEEE Computer Society.

[BH05] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. *SIGMETRICS Performance Evaluation Review*, 33(1):169–180, 2005.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, (PACT '08), pages 72–81, New York, NY, USA, 2008. ACM.

[BS72] R. H. Bartels and G. W. Stewart. Algorithm 432: Solution of the matrix equation ax + xb = c. *Communications of the ACM*, pages 820–826, 1972.

[BS96] Frank Bellosa and Martin Steckermeier. The performance implications of locality information usage in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 37:113–121, August 1996.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.

[CDC+99] W. Carlson Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical report, CCS-TR-99-157, George Mason University, Mai 1999.

[CGKS05] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, (HPCA '05), pages 340–351, Washington, DC, USA, 2005. IEEE Computer Society.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[CJ06] Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, Washington, DC, USA, 2006. IEEE Computer Society.

[CP03] Calin Cascaval and David A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of the 17th annual international conference on Supercomputing (CDROM)*, (ICS '03), pages 150–159, New York, NY, USA, 2003. ACM.

[CPHL01] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. Exact analysis of the cache behavior of nested loops. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, (PLDI '01), pages 286–297, New York, NY, USA, 2001. ACM.

[CSG98] David Culler, J.P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[DC09] Chen Ding and Trishul Chilimbi. A composable model for analyzing locality of multi-threaded programs. Technical report, MSR-TR-2009-107, Microsoft Research, 2009.

[DM03] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. Technical report, RedHat, Inc, 2003.

[DZ01] Chen Ding and Y Zhong. Reuse distance analysis. Technical report, UR-CS-TR-741, University of Rochester, Rochester, NY, USA, 2001.

[DZ03] Chen Ding and Yutao Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, (PLDI '03), pages 245–257, New York, NY, USA, 2003. ACM.

[Edm65] Jack Edmonds. Maximum matching and a polyhedron with 0-1 vertices. *Journal of Research of the National Bureau of Standards*, 69-B(1-22):125–130, January-June 1965.

[Era04] Eranian, Stephane. The perfmon2 interface specification. Technical report, HPL-2004-200R1, Hewlett-Packard Laboratory, February 2004. http://www.hpl.hp.com/techreports/2004/HPL-2004-200R1.html.

[Fah97] Thomas Fahringer. Estimating cache performance for sequential and data parallel programs. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, (HPCN Europe '97), pages 840–849, London, UK, UK, 1997. Springer-Verlag.

[FDZ99] Basilio B. Fraguela, Ramon Doallo, and Emilio L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, (PACT '99), pages 221–, Washington, DC, USA, 1999. IEEE Computer Society.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, (PLDI '98), pages 212–223, New York, NY, USA, 1998. ACM.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, (OOPSLA '07), pages 57–76, New York, NY, USA, 2007. ACM.

[GMM99] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):703–746, July 1999.

[GNU] GNU. GNU Pth: The GNU Portable Threads. http://www.gnu.org/s/pth/.

[Gri09] Grigori Fursin and Olivier Temam. Collective Optimization. In *the 4th International Conference on High Performance and Embedded Architectures and Compilers (HIPEAC)*, volume 5409 of *Lecture Notes in Computer Science*, pages 34–49. Springer, 2009.

[HKA+01] Christopher J. Hughes, Praful Kaul, Sarita V. Adve, Rohit Jain, Chanik Park, and Jayanth Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *Proceedings of the 28th annual international*

symposium on Computer architecture, (ISCA '01), pages 254–265, New York, NY, USA, 2001. ACM.

[Hug09] Hugh Leather and Michael O'Boyle and Bruce Worton. Raced Profiles: Efficient Selection of Competing Compiler Optimizations. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '09)*. ACM SIGPLAN/SIGBED, June 2009.

[Int] Intel. Intel Threading Building Blocks (TBB) for Open Source. http://threadingbuildingblocks.org/.

[JFY99] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, NAS-99-011, NASA Ames Research Center, October 1999. http://www.nas.nasa.gov/Resources/Software/npb.html.

[JM10] Emmanuel Jeannot and Guillaume Mercier. Near-optimal placement of MPI processes on hierarchical NUMA architectures. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, (Euro-Par'10), pages 199–210, Berlin, Heidelberg, 2010. Springer-Verlag.

[Joh88] Eric E. Johnson. Completing an MIMD multiprocessor taxonomy. *SIGARCH Computer Architercture News*, 16:44–47, June 1988.

[Joh02] John L. Hennessy and David A. Patterson and David Goldberg . *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002. ISBN-13: 978-1558605961.

[JZTS10] Yunlian Jiang, Eddy Z. Zhang, Kai Tian, and Xipeng Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *19th International Conference on Compiler Construction (CC)*, volume 6011 of *Lecture Notes in Computer Science*, pages 264–282. Springer, 2010.

[KBH+08] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28:54–66, May 2008.

[KFA08] Vahid Kazempour, Alexandra Fedorova, and Pouya Alagheband. Performance implications of cache affinity on multicore processors. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, (Euro-Par '08), pages 151–161, Berlin, Heidelberg, 2008. Springer-Verlag.

[KK98a] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, December 1998.

[KK98b] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, (Supercomputing '98), pages 1–13, Washington, DC, USA, 1998. IEEE Computer Society.

[KK98c] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, January 1998.

[Kle05] Andi Kleen. A NUMA API for Linux. Technical report, Novell-4621437, Novell, April 2005.

[KMN+09] Mahmut Kandemir, Sai Prashanth Muralidhara, Sri Hari Krishna Narayanan, Yuanrui Zhang, and Ozcan Ozturk. Optimizing shared cache behavior of chip multiprocessors. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 505–516, New York, NY, USA, 2009. ACM.

[KOWT11] Tobias Klug, Michael Ott, Josef Weidendorfer, and Carsten Trinitis. autopin: automated optimization of thread-to-core pinning on multicore systems. In *Transactions on high-performance embedded architectures and compilers III*, pages 219–235. Springer-Verlag, Berlin, Heidelberg, 2011.

[KSP09] Thomas Karcher, Christoph Schaefer, and Victor Pankratius. Auto-tuning support for manycore applications: perspectives for operating systems and compilers. *ACM SIGOPS Operating System Review*, 43(2):96–97, 2009.

[Kuh55] Harold W. Kuhn. The Hungarian Method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[KYM+10] Mahmut Kandemir, Taylan Yemliha, SaiPrashanth Muralidhara, Shekhar Srikantaiah, Mary Jane Irwin, and Yuanrui Zhnag. Cache topology aware computation mapping for multicores. *SIGPLAN Not.*, 45(6):74–85, 2010.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, (PLDI '05), pages 190–200, New York, NY, USA, 2005. ACM.

[LLD+08] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *14th International Symposium on High Performance Computer Architecture*, (HPCA '08), pages 367–378. IEEE Computer Society, 2008.

[LR95] Peter Lancaster and Leiba Rodman. *Algebraic Riccati equations*. Oxford University Press, 1995.

[LWRC10] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *Proceedings of the 37th annual international symposium on Computer architecture*, (ISCA '10), pages 270–279, New York, NY, USA, 2010. ACM.

[MCT96] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transaction on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, July 1996.

[MDHS09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th international conference on Architectural support for programming languages*

and operating systems, (ASPLOS '09), pages 265–276, New York, NY, USA, 2009. ACM.

[MGST70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, June 1970.

[MMSW02] Bernd Mohr, Allen D. Malony, Sameer Shende, and Felix Wolf. Design and prototype of a performance tool interface for openmp. *The Journal of Supercomputing*, 23:105–128, August 2002.

[Moo75] Gordon E. Moore. Progress in digital integrated electronics. In *International Electron Devices Meeting*, volume 21, pages 11–13. IEEE, 1975.

[Moo02] Shirley V. Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In *Proceedings of the International Conference on Computational Science-Part II*, (ICCS '02), pages 904–912, London, UK, UK, 2002. Springer-Verlag.

[MPI] MPI forum. The Message Passing Interface Standard. http://www.mpi-forum.org/.

[MTB10] Abdelhafid Mazouz, Sid-Ahmed-Ali Touati, and Denis Barthou. Measuring and Analysing the Variations of Program Execution Times on Multicore Platforms: Case Study. Technical report, HAL-inria-00514548, University of Versailles Saint-Quentin en Yvelines, 2010.

[MTM10] Jaydeep Marathe, Vivek Thakkar, and Frank Mueller. Feedback-directed page placement for ccNUMA via hardware-generated memory traces. *Journal of Parallel and Distributed Computing*, 70:1204–1219, December 2010.

[Net94] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation.* PhD thesis, University of Cambridge, November 2004, 1994.

[NR98] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, (PLDI '07), pages 89–100, New York, NY, USA, 2007. ACM.

[oBFG+10] Franҫis Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal on Parallel Programming, Special Issue on OpenMP; Guest Editors: Matthias S. Muller and Eduard Ayguadé*, 38(5):418–439, 2010.

[Per] Perfctr. http://user.it.uu.se/ mikpe/linux/perfctr/.

[PJN08] Marc Pérache, Hervé Jourdren, and Raymond Namyst. MPC: A unified parallel runtime for clusters of numa machines. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, (Euro-Par '08), pages 78–88, Berlin, Heidelberg, 2008. Springer-Verlag.

[PKB+91] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multi-thread architecture. In *In Proceedings of the Winter 1991 USENIX Conference*, pages 65–80, 1991.

[QP06] Moinuddin K. Qureshi and Yale N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, Washington, DC, USA, 2006. IEEE Computer Society.

[Raj91] Raj Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modelling*. John Wiley and Sons, New York, 1991.

[RLT06] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven CMP cache management. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, (PACT '06), pages 2–12, New York, NY, USA, 2006. ACM.

[SDR02] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, (HPCA '02), page 117, Washington, DC, USA, 2002. IEEE Computer Society.

[SKM+06] Srinivas Sridharan, Brett Keck, Richard Murphy, Surendar Ch, and Peter Kogge. Thread migration to improve synchronization performance. In *In Workshop on Operating System Interference in High Performance Applications*, 2006.

[SKP10] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, (PACT '10), pages 53–64, New York, NY, USA, 2010. ACM.

[SMD07] Fengguang Song, Shirley Moore, and Jack Dongarra. Feedback-directed thread scheduling with memory considerations. In *Proceedings of the 16th international symposium on High performance distributed computing*, (HPDC '07), pages 97–106, New York, NY, USA, 2007. ACM.

[SMD09] Fengguang Song, Shirley Moore, and Jack Dongarra. Analytical modeling and optimization for affinity based thread scheduling on multicore systems. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing (CLUSTER), August 31 - September 4, 2009, New Orleans, Louisiana, USA*, pages 1–10. IEEE, 2009.

[SPP10] Derek L. Schuff, Benjamin S. Parsons, and Vijay S. Pai. Multicore-aware reuse distance analysis. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS) Workshops*, pages 1–8. IEEE, 2010.

[SRD04] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28:7–26, April 2004.

[Sta06] Standard Performance Evaluation Corporation. SPEC CPU. http://www.spec.org/, 2006.

[STaMB10] Dirk Schmidl, Christian Terboven, Dieter an Mey, and H. Martin Bucker. Binding nested openmp programs on hierarchical memory architectures. In *International Workshop on OpenMP*, pages 29–42, 2010.

[Syl84] J.J Sylvester. Sur l'equations en matrices px=xq. *Comptes Rendus des Séances de l'Académie des Sciences. Paris*, pages 67–71,115–116, 1884.

[TaMS+08] Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. Data and thread affinity in openmp programs. In *Proceedings of the 2008 workshop on Memory access on future processors*, (MAW '08), pages 377–384, New York, NY, USA, 2008. ACM.

[TAS07] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, (EuroSys '07), pages 47–58, New York, NY, USA, 2007. ACM.

[TDF90] George Taylor, Peter Davies, and Michael Farmwald. The TLB slice – a low-cost high-speed address translation mechanism. In *Proceedings of the 17th annual international symposium on Computer Architecture*, (ISCA '90), pages 355–363, New York, NY, USA, 1990. ACM.

[Thea] The Open Group Base Specifications. POSIX. http://pubs.opengroup.org/onlinepubs/009695399/.

[Theb] The OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. http://www.openmp.org/.

[TMV+11] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Proceeding of the 38th annual international symposium on Computer architecture*, (ISCA '11), pages 283–294, New York, NY, USA, 2011. ACM.

[TWB10] Sid-Ahmed-Ali Touati, Julien Worms, and Sebastien Briais. The Speedup-Test. Technical report, HAL-inria-00443839, University of Versailles Saint-Quentin en Yvelines, January 2010. http://hal.archives-ouvertes.fr/inria-00443839.

[TWB12] Sid-Ahmed-Ali Touati, Julien Worms, and Sébastien Briais. The Speedup-Test: A Statistical Methodology for Program Speedup Analysis and Computation. *To appear in the Journal of Concurrency and Computation: Practice and Experience*, 2012.

[VX02] Xavier Vera and Jingling Xue. Let's study whole-program cache behaviour analytically. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, (HPCA '02), pages 175–, Washington, DC, USA, 2002. IEEE Computer Society.

[Wei98] Boris Weissman. Performance counters and state sharing annotations: a unified approach to thread locality. In *Proceedings of the 8th international conference on Architectural support for programming languages and operating systems*, (ASPLOS '98), pages 127–138, New York, NY, USA, 1998. ACM.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, (PLDI '91), pages 30–44, New York, NY, USA, 1991. ACM.

[WM08] Vincent M. Weaver and Sally A. McKee. Can hardware performance counters be trusted? In *Proceedings of the 4th International Symposium on Workload Characterization (IISWC 2008)*, pages 141–150. IEEE Computer Society, September 14-16 2008.

[WY11] Meng-Ju Wu and Donald Yeung. Coherent profiles: Enabling efficient reuse distance analysis of multicore scaling for loop-based parallel programs. In *Proceedings of the 20th international conference on Parallel architectures and compilation techniques*, (PACT '11), New York, NY, USA, 2011. ACM.

[XL08] Yuejian Xie and Gabriel H. Loh. Dynamic classification of program memory behaviors in CMPs. In *Proceedings of CMP-MSI, held in conjunction with ISCA-35*, 2008.

[YSP+98] Katherine A. Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul N. Hilfinger, Susan L. Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A high-performance java dialect. *Concurrency - Practice and Experience*, 10(11-13):825–836, 1998.

[ZBF10] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems*, (ASPLOS '10), pages 129–142, New York, NY, USA, 2010. ACM.

[ZDS09] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, (EuroSys '09), pages 89–102, New York, NY, USA, 2009. ACM.

[ZJH09] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Accuracy of performance counter measurements. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, (ISPASS '09), pages 23–32. IEEE Computer Society, April 26-28 2009.

[ZJS10] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, (PPoPP '10), pages 203–212, New York, NY, USA, 2010. ACM.

[ZKY11] Yuanrui Zhang, Mahmut Kandemir, and Taylan Yemliha. Studying inter-core data reuse in multicores. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, (SIGMETRICS '11), pages 25–36, New York, NY, USA, 2011. ACM.

[ZSD09] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):1–39, 2009.