Universitat Autònoma de Barcelona

Escola d'Enginyeria

Departament d'Arquitectura de Computadors i Sistemes Operatius

Modeling performance degradation in OpenMP memory bound applications on multicore multisocket systems.

Thesis submitted by **César Allande Álvarez** for the degree of philosophae Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dr. Eduardo César Galobardes and Dr. Josep Jorba Esteve, developed at the Computer Architectures and Operating Systems department, PhD in High Performance Computing.

Barcelona, November 2015

Modeling performance degradation in OpenMP memory bound applications on multicore multisocket systems.

Thesis submitted by **César Allande Álvarez** for the degree of Philosophiae Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dr. Eduardo César Galobardes and Dr. Josep Jorba Esteve, at the Computer Architecture and Operating Systems Department, Ph.D in High performance Computing.

Supervisors

Dr. Eduardo César Galobardes

Dr. Josep Jorba Esteve

Barcelona, November 2015

Dedications

To my family.
To my friends.

Had I the heavens' embroidered cloths,
Enwrought with golden and silver light,
The blue and the dim and the dark cloths
Of night and light and the half-light,
I would spread the cloths under your feet:
But I, being poor, have only my dreams;
I have spread my dreams under your feet;
Tread softly because you tread on my dreams.

The cloths of Heaven – **William Butler Yeats**

Acknowledgements

I would like to express sincere thanks to my advisors Dr. Eduardo César and Dr. Josep Jorba for their support during the undertaking of this research.

I would like to thank all the members of the examining committe for their thoughful comments and suggestions in this dissertation.

I am indebted to Dr. Karl Fuerlinger of Munich Network Mangement Team for his hospitality, and participation along my research stay at Ludwig-Maximilians-Universität München. The completion of this dissertation would not have been possible without his encouragement and guidance. Furthermore, I would like to extend my gratitude to Prof. Dr. Dieter Kranzlmüller and all the members of the MNM team.

I would also like to thank all the people in the Leibniz-Rechenzentrum for their welcome and support. Sincerely in debt with Dr. Reinhold Bader and Dra. Sandra Mendez for the assistance, prolific discussions and feedback along my stay.

I would like to thank all the people at the Barcelona Supercomputing Center and my fellows at the Riding on Moore's Law project (RoMoL) for the participation, support and guidance, specially to prof. Dr. Mateo Valero, Dr. Miquel Moretó, Dr. Marc Casas, Luc Jaulmes, Vladimir Dimic, Xubin Tan, Dimitrios Chasapis, Emilio Castillo, and Lluc Alvarez.

I also want to thank all the members and staff at the Computer Architectures and Operating Systemps department at the UAB, especially to Dolores Rexachs, Emilio Luque, Joan Sorribes, Sandra Méndez, Aprigio Bezerra, Eduardo César Cabrera, Hugo Meyer, Arindam Choudhury, Marcela Castro, Tharso Souza, Roberto Solar, Joao Gramacho, Alex Guevara, Julio César Vizcaíno, Alejandro Chacón, Tomás Artés, Gemma Sanjuan, Gemma Roque, Javier Navarro, Daniel Ruiz, and Manuel Brugnoli (r.i.p.).

Abstract

The evolution of multicore processors has completely changed the evolution of current HPC systems. The multicore architectures were mainly designed to avoid three design walls, instruction level parallelism, power wall, and finally the memory wall. The last because of the increasingly gap between processor and memory speeds.

Performance of memory intensive applications executed on multi-core multi-socket environments is closely related to the utilization of shared resources in the memory hierarchy. The shared resources utilization can lead to a significant performance degradation. The exploration of different thread affinity configurations allows the selection of a proper configuration that balances the performance improvement obtained by increasing parallelism with the performance degradation due to memory contention.

The main contributions of this thesis is the definition of a methodology for developing dynamic tuning strategies in multicore multisocket environment which has provided the definition of two performance models for memory intensive applications.

The first performance model, based on runtime characterization, estimates the execution time for different configurations of threads and thread distributions in a multicore multisocket system. To do that, the model requires a runtime characterization from the exhaustive execution on a single socket to determine the memory contention.

The second performance model, based on pre-characterization of the application, estimates at runtime the impact of the memory contention for concurrent executions based on profiling the memory footprint from traces of the application using small workloads.

Keywords: Performance Model, Multicore, Multisocket, Memory Contention, OpenMP Applications, Memory Footprint

Resumen

La evolución de los procesadores multicore ha cambiado completamente la evolución de los actuales sistemas de HPC. Las arquitecturas multicore han sido diseñadas principalmente para evitar tres barreras de diseño, el paralelismo a nivel de instrucción, el consumo energético y la contención de memoria. La última debido a la creciente diferencia de velocidad entre el procesador y la memoria.

El rendimiento de aplicaciones intensivas en memoria ejecutadas en entornos multicore multisocket está relacionado directamente a la utilización de los recursos compartidos en la jerarquía de memoria. La utilización de recursos compartidos puede llevar a una degradación de rendimiento significativa. La exploración de diferentes configuraciones de afinidad de threads permite la selección de configuraciones que pueden llegar a equilibrar la mejora de rendimiento obtenido por el incremento de paralelismo con la degradación debida a la contención de memoria.

La principales contribuciones de esta tesis es la definición de una metodología para el desarrollo de estrategias de sintonización en entornos multicore multisocket que ha proporcionado la definición de dos modelos de rendimiento para aplicaciones intensivas en memoria.

El primer modelo de rendimiento, basado en una caracterización en tiempo de ejecución, estima el tiempo de ejecución para diferentes configuraciones de numero y distribución de threads para entornos multicore multisocket. Para ello, el modelo requiere de una caracterización exhaustiva en tiempo de ejecución sobre un solo procesador con el objetivo de determinar la contención de memoria.

El segundo modelo de rendimiento, basado en la pre-caracterización de la aplicación, estima en tiempo de ejecución el impacto de la contención de memoria para ejecuciones concurrentes basado en un perfil del memory footprint extraído de trazas de la aplicación ejecutada con pequeñas cargas de trabajo.

Palabras clave: Modelo de Rendimiento, Multicore, Multisocket, Contención en memoria, aplicaciones OpenMP, Perfil de memoria

Resum

L'evolució dels processadors multicore ha canviat completament l'evolució dels actuals sistemes de HPC. Les arquitectures multicore han estat dissenyades principalment per evitar tres barreres de disseny: el paral·lelisme a escala d'instrucció, el consum energètic i la contenció a memòria. La darrera és deguda a la creixent diferència de velocitat entre el processador i la memòria.

Les prestacions de les aplicacions intensives a memòria executades en entorns multicore multisocket estan directament relacionades a la utilització dels recursos compartits a la jerarquia de memòria. La utilització dels recursos compartits pot portar a una degradació de les prestacions significativa. L'exploració de diferents configuracions d'afinitat de threads permet la selecció de configuracions que poden arribar a equilibrar la millora de prestacions obtinguda deguda a l'increment del paral·lelisme amb la degradació deguda a la contenció a memòria.

Les principals contribucions d'aquesta tesi és la definició d'una metodologia pel desenvolupament d'estratègies de sintonització en entorns multicore multisocket que ha proporcionat la definició de dos models de rendiment per aplicacions intensives a memòria.

El primer model de rendiment, basat en una caracterització en temps d'execució, estima el temps d'execució per diferents configuracions de número i distribució de threads en entorns multicore multisocket. Per aquesta finalitat, el model requereix una caracterització exhaustiva en temps d'execució en un únic processador amb l'objectiu de determinar la contenció a memòria.

El segon model de rendiment, basat en la pre-caracterització de l'aplicació, estima el temps d'execució i l'impacte de la contenció a memòria per execucions concurrents basat en el perfil del memory footprint extret de traces de la mateixa aplicació executada amb petites càrregues de treball.

Paraules clau: Model de Rendimient, Multicore, Multisocket, Contenció a memoria, aplicacions OpenMP, Perfil de memòria

Contents

1 Introduction **1**

1.1 Context . . . 2
1.2 Motivation . . . 4
1.3 Objectives . . . 5
1.4 Contribution . . . 6
1.5 Thesis outline . . . 7

2 Background **9**

2.1 High performance computing . . . 10
2.2 Multicore architectures . . . 12
2.2.1 State of the art multicore processors . . . 13
2.3 OpenMP specification . . . 19
2.4 System characterization, monitoring, and instrumentation tools . . . 24
2.4.1 System characterization by benchmarking . . . 25
2.4.2 Monitoring tools . . . 27
2.4.3 Performance analysis tools . . . 28
2.4.4 Dynamic Instrumentation . . . 29
2.5 Related work . . . 31
2.6 Summary . . . 32

3 Methodology for developing tuning strategies for OpenMP applications **35**

3.1 Objective . . . 36
3.2 Methodology . . . 37
3.2.1 System characterization . . . 38
3.2.2 Analysis of performance factors . . . 41
3.2.3 Modeling performance and defining tuning strategies . . . 45
3.2.4 Evaluating the impact . . . 46

3.3 Summary 46

4 Methodology application to a case study **49**
4.1 Methodology application on NAS Parallel Benchmarks 50
4.2 Context analysis for the identification of performance factors 50
4.2.1 System Characterization 50
4.2.2 Analysis of performance factors 61
4.3 Evaluating a strategy for tuning the number of threads 64
4.3.1 Context 65
4.3.2 Modeling performance and defining tuning strategies 69
4.3.3 Evaluating the impact 70
4.3.4 Applying the dynamic tuning strategy 71
4.4 Summary 72

5 Performance model based on runtime characterization **75**
5.1 Introduction 76
5.2 Objective 76
5.3 Related work 77
5.4 Performance Model proposal 78
5.4.1 Defining the performance model 79
5.5 Experimental validation 83
5.5.1 Applying the model for the SP application on the T7500 system. . . 85
5.5.2 Selecting a configuration for SP and MG benchmarks on FatNode . 86
5.5.3 Exploration of the affinity configurations. 88
5.6 Summary 90

6 Performance model based on profiling the memory footprint **91**
6.1 Introduction 92
6.2 Objective 92
6.3 Related work 93
6.4 Methodology for selecting a configuration to avoid memory contention . . . 94
6.4.1 Trace generation and characterization of iteration footprint 95
6.4.2 Estimation of memory footprint at runtime 97
6.4.3 Estimation of execution time for all configurations at runtime 97
6.5 Experimental Validation 100

6.5.1 Obtaining memory footprints and execution times 100
6.5.2 Estimating the best configuration at runtime 105
6.5.3 Experimental Results 107
6.6 Summary 110

7 Conclusions **111**
7.1 Conclusions 112
7.2 Future Work 114
7.3 List of publications 115
7.4 Acknowledgements 117

Bibliography **119**

List of Figures

1.1 Evolution of number of cores in a socket of the top 500 supercomputers from Top500 list Juny 2006 to 2015. 4

2.1 Power8 processor from IBM. A processor based on multi-level cache hierarchy of 3 levels in processors, and an external L4 cache. This processor provides SMT of 8 hardware threads per core. cores with L3 cache partitioning. . . . 16
2.2 NUMA environment with 2 processors containing 2 NUMA clusters each. . 16
2.3 big.LITTLE processor from ARM. Heterogeneous processor with a set of 4 faster Cortex-A15 cores for high performance and a set of 4 Cortex-A7 in order cores for power efficiency. 17
2.4 GM107 Maxwell processor from NVIDIA, a multi/many-core coprocessor device. 17
2.5 Coprocessor execution flow on a CUDA device. 18
2.6 Knights Landing processor from Intel, with a 2D mesh tile interconnection of 36 tiles, were every tile contains 2 cores with SMT of 8 hardware threads per each. 18

3.1 Methodology for generating dynamic tuning strategies in multicore systems 38
3.2 Context characteristics development template 42
3.3 Performance problems on OpenMP applications 43

4.1 Processor architecture on System FatNode 51
4.2 Evaluation of memory latencies on FatNode system with `lat_mem_rd` 51
4.3 Processor architecture on System T7500 52
4.4 Evaluation of memory latencies on t7500 system with `lat_mem_rd` 53
4.5 Scalability analysis on EP benchmark 54
4.6 Scalability analysis on CG benchmark 55
4.7 Scalability analysis on FT benchmark 56
4.8 Scalability analysis on MG benchmark 57

4.9 Scalability analysis on BT benchmark 58
4.10 Scalability analysis on LU benchmark 59
4.11 Scalability analysis on SP benchmark 60
4.12 Total cache misses and execution time on parallel region x_solve for SP.C . 62
4.13 SP.C x_solve – Heatmap of hardware counter Total Instructions 62
4.14 SP.C x_solve – Heatmap of hardware counter Total Cache Misses 63
4.15 Scalability analysis for SP Classes B and C for systems Sysα and Sysβ . . 66
4.16 System Sysβ (4 threads) - 1 SP.C iteration 67
4.17 System Sysβ (5 threads) - 1 SP.C iteration 67
4.18 Methodology to select the configuration providing the minimum execution time based on an exhaustive runtime characterization on the first iterations of the application. 69

5.1 Methodology for the selection of the number of threads and its affinity distribution. 79
5.2 Evaluation of execution time between estimated boundaries. 87
5.3 SP_C_xsolve local allocation 88
5.4 MG_C_R0011 local allocation 89

6.1 Methodology to select a configuration that avoid memory contention based on the analysis of the concurrent memory footprint in LLC 94
6.2 Full memory trace visualization for x_solve parallel region on SP Class W . . 101
6.3 Detailed view of memory trace for x_solve parallel region on SP Class W . . 101
6.4 Analysis of concurrent footprint for x_solve parallel region for all workload classes in an architecture with LLC of 20MB 104
6.5 Comparison of measured and the estimated sequential iteration time using a linear regression interpolation from classes S, W and A. 106
6.6 Estimation of iteration time degradation on x_solve parallel region on SP.C . 106
6.7 Comparison of model estimated execution times against real execution times on different architectures (MN3, SuperMUC Fat and Thin nodes) for parallel regions Copy and Add from the stream benchmark. 108
6.8 Comparison of model estimated execution times against real execution times for parallel regions x_solve for the SP benchmark using distribution policies compact (AFF0) and scattered (AFF1). 109

List of Tables

4.1 Sysβ Class C execution time (sec.) and cumulative percentage (relative to total time T_{ref}) of use for the weightiest parallel regions (x,y and z_solve, and rhs). 68
4.2 x_solve parallel region on Sysβ Class C for one iteration execution. Where T_{it_n} is the time for n-iteration and T_{ref} is the measured time for 400 iterations. 68
4.3 Execution time (sec.) for the dynamic tuning strategy and execution without tuning for classes B and C. The tuning strategy uses 5% of total iterations for the characterization stage. 72

5.1 Table of parameters used to estimate the execution time of N threads for a given configuration. 80
5.2 System hardware characteristics at node level. 84
5.3 T7500 system. Input data for x_solve parallel region from SP benchmark class C. 85
5.4 T7500 system. SP class C with affinity $AFF1$. Estimation and evaluation of TCM for parallel region x_solve. 86
5.5 Selection of configuration for SP and MG benchmarks 87
5.6 Execution time for selected configuration and speedups. 90

6.1 Description of system architectures for performance evaluation 100
6.2 Preprocessed information from traces of small classes. This information is required to estimate performance at runtime. The profile information of the memory intensive parallel region x_solve specifies the per cluster memory footprint for one iteration. Besides, inputs for the performance model such as the cumulative iteration footprint ($iterFootprint$) and the iteration serial execution time, are shown. 103
6.3 Stream Benchmark configuration, and the iteration footprint estimation per operation 103

6.4 Estimation of memory footprint for a concurrent execution of x_solve parallel region, where $^{\dagger}first\_contention$ on MN3 and Thin nodes, and $^{*}first\_contention$ for system Fat node 104
6.5 Estimation of serial iteration time in seconds on MN3 with 20MB LLC. The highlighted cells refer to information obtained on the characterization phase. Serial Estimation time (Est.) is obtained from an interpolation of a linear regression function between footprints and measured times for classes S, W, and A 105
6.6 Speedup evaluation of the selected configuration compared with best configuration. 110

1 Introduction

"Changes and progress very rarely are gifts from above. They come out of struggles from below."

- **Noam Chomsky**

In this chapter we present a general overview of High Performance Computing (HPC) and the scope of this thesis. This work is focused on performance analysis and performance modeling of OpenMP memory intensive applications executed in multicore systems. This chapter introduces motivation, the objectives we have defined, and the contributions we have provided. Finally, we present the thesis outline.

1.1 Context

Performance of parallel applications in High Performance Computing (HPC) is expected to increase proportionally to the number of used resources. In order to do that, HPC systems take benefit of the interconnection of multiple nodes by managing a coordinated parallel execution. Nowadays, due to the increasing integration capacity of components, parallelism takes benefit of multicore processors. Performance on current processors does not depend only on the processor frequency, but on their number of cores.

However, multicore architectures have such heterogeneity, that programming frameworks and/or languages have to manage parallelism and take into consideration many possible configurations, going from multicore processors based on a shared memory hierarchy to network based interconnection topologies such as the mesh on-chip network on Tilera[1] with 64 to 72 cores, or the upcoming Intel's Xeon Phi Knights Landing processor with 72 Silvermont-based cores, both processors with a 2D mesh interconnection. Furthermore, current designs are decoupling from a very powerful general purpose multicore processor to approaches based on co-processing and accelerators such as Intel's Many Integrated Core architecture [2] or the NVIDIA's CUDA General-Purpose computing on Graphics Processing units (GPGPU).

Therefore, to improve the benefit of using HPC systems, applications use one or more programming models to deal with the underlying architecture. The Message Passing Interface (MPI) has been designed for distributed memory systems, but it is able to take benefit of shared memory system by coordinating processes at the same node. However, multicore systems are usually managed by frameworks implementing the shared memory programming model, such as OpenMP [3], OmpSs [12], Intel Threading Building Blocks (Intel TBB) [4], OpenACC [5], or Cilk [6].

Consequently, to obtain the full potential of a parallel system it is necessary to deal with the programming frameworks. We rely on them to effectively manage and coordinate the parallel execution. Nevertheless, the execution of parallel application does not always achieve the expected performance.

The main reason for this problem is that modern multicore systems are designed to integrate cores within a processor sharing some module units such as last level cache, memory controllers, interconnect, or prefetching hardware, and the management of shared resources and coordination of processing units can limit the overall performance of applications. On multicore architectures, the access to the memory hierarchy is possibly the most important limiting factor, specially for memory intensive applications [7].

A way for improving the application's performance is to dynamically adjust the parallel execution configuration to the characteristics of the hardware architecture. This can be done by identifying a performance bottleneck and dynamically tuning the management of the parallelism to meet the application's requirements.

In those Shared memory APIs, a runtime library manages the parallelism, and that library can be adapted for deploying performance strategies. To adjust the runtime execution, it is possible to implement a performance tuning functionality on an existent library, using an interposition library, or using dynamic instrumentation tools, such as Pin[8] or Dyninst[9] in order to tune the parameters of an existent parallel library.

OpenMP is one of the most widely spread Application Program Interface (API) for multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP specification defines a collection of compiler directives, library routines, and environment variables for shared-memory parallelism programs. There are several OpenMP-compliant implementations and it is extensively used on HPC systems. The OpenMP implementations provide a interface between the application and the runtime library responsible of managing the thread parallelism. Using an instrumentation tool, it is possible to intercept the API library calls to modify the applications/library runtime behaviour.

It is possible to use a interposition library to tune significant parameters of the OpenMP runtime library in order to modify the number of threads, the thread mapping policy, or the workload scheduling for a parallel region, among others. We consider these as the tuning parameters to be adjusted by a tuning tool.

A tuning tool must contain three main components, the monitoring points, the performance problem identification and a tuning strategy. We use analytical models to identify a performance problem and evaluate its impact on the application performance, and when necessary apply a specific tuning strategy.

In this work, we define a methodology for providing all the elements required to develop a tuning tool to supervise the execution of OpenMP scientific applications in HPC systems. To do that, we have described a significant performance problem based on memory contention, which has been analytically modelled, describing the monitoring points, its impact on performance, and estimate the parameters that can be tuned in the parallel runtime library providing a solution which avoids or minimizes the performance problem.

In this chapter, we present a general overview of the thesis, by introducing the following subsections: the motivation of this research, its general and specific objectives, and its contributions. Finally, the last subsection introduces the thesis outline.

1.2 Motivation

The evolution of multicore processors has completely changed the evolution of current HPC systems. Figure 1.1 shows the number of cores per processors in the top 500 HPC supercomputers [10].

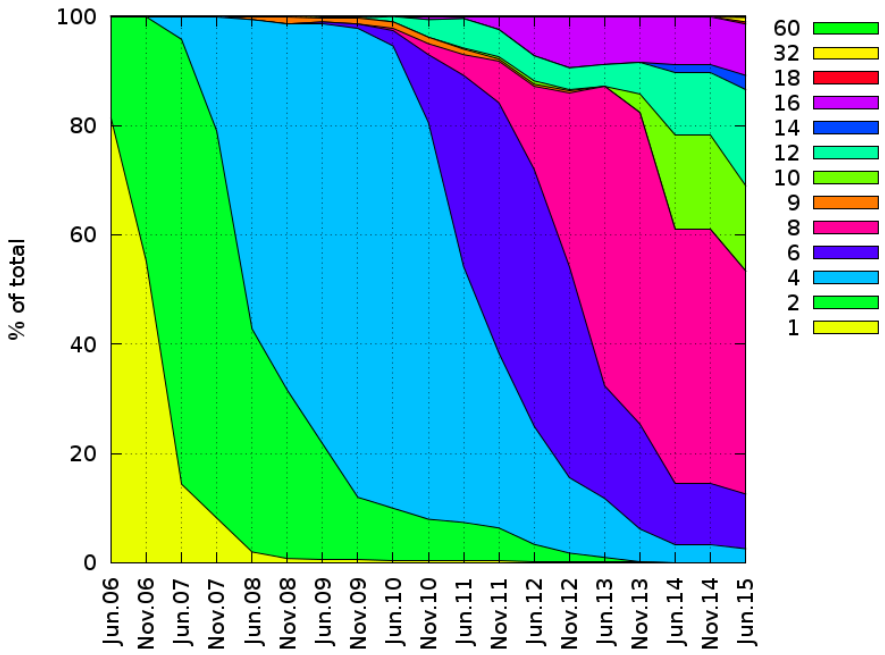


Figure 1.1: Evolution of number of cores in a socket of the top 500 supercomputers from Top500 list Juny 2006 to 2015.

It can be observed that the processing units integrated in such systems tend to provide more parallelism within processors year after year. In almost ten years the dominant number of cores per processors has changed from mono-processors to 8 core processors. Furthermore, nowadays the tendency is to integrate coprocessors such as the Intel Xeon Phi providing up to 60 cores using x86 instruction set.

Despite the sharp increase on the number of resources on HPC systems, the benefit of using multicore systems is usually not for free. The applications require to be adapted to use these parallel resources manly by exposing their parallelism through parallel programming models. Eventhough applications express well balanced data partitioning and full parallelism on their codes, the performance is not always linear to the number of resources, as expected. There are different factors limiting their scalability, which are mainly related to the design of the multicore architectures.

Multicore processors are a compact design of processing units encapsulated on a single chip. In order to improve integration there are some modules which are shared among cores. To do that, there are different protocols managing the coordinated access to shared elements to grant its usability and data integrity. That shared utilization of resources can lead to application bottlenecks which can slowdown performance.

The performance analysis is a procedure for quantifying in a valid and consistent manner the key elements of performance. Performance analysis of applications executed in multicore environment can provide the insights of performance problems and consequently the key factors to consider in order to find a solution.

A performance model is a formal way of describing the performance key elements and their relations. The performance model lead us to pre-empt performance problems by exploring different values on its parameters. This can be used to identify the proper conditions for the execution context.

All of this, can be summarized in the motivation of this work; ***expose the intrinsics of performance problems to relate application characteristics to the hardware architecture in the context of parallel applications executed in multicore systems*** .

1.3 Objectives

The ultimate goal of this work is to identify performance problems in the context of OpenMP applications executed in multi-core multi-sockets environments and relate the key elements within the performance problem through a performance model that considers the application characteristics and the hardware architecture.

With the aim of achieving this objective we have developed the following specific objectives:

- Definition of a work methodology to identify performance problems on OpenMP applications by evaluating its impact on performance, analysing relevant observable runtime metrics, and describing possible tuning strategies to be performed at runtime. This methodology can be summarized in the following steps:
  - Context characterization.
  - Identifying performance problems.
  - Evaluation of the impact on performance.

  - Evaluation of tuning parameters
  - Take benefit of iterative patterns of the application to apply a tuning strategy.

- Through the development of the methodology, identify a relevant performance factors and define a performance model providing a performance estimation at runtime and the tuning parameters to be modified in order to improve performance.

- Defining a performance model based on performance degradation on memory bound applications executed in multicore multisocket systems. Following this, we aimed for a performance model to perform a runtime only characterization, and following this, a performance model based on pre-characterization of the application in order to improve the automation of the tuning strategy. This can be summarize as the following specific objectives:

  - A performance model based on LLC misses characterization on a single socket.
  - A performance model based on spatial access pattern characterization using a memory profile of the application.

1.4 Contribution

The contributions of this work are focused on achieving the ultimate goal presented in the previous section. To this end, we have designed and developed a methodology to analyze performance problems in OpenMP applications, and developed two performance models for memory intensive applications to identify memory contention and to provide a configuration of number of threads and the thread distribution in a multicore multisocket system.

Specifically, this thesis presents the following contributions:

- A methodology for defining dynamic tuning strategies in multicore systems; This methodology has been defined in order to structure the workflow with the aim of identifying significant performance problems in the context of HPC and to explore the strategies required to tune the runtime system in order to improve the application performance. The application of this methodology has allowed the identification of an effective way to approach strategies to minimize performance degradation on memory bound applications. Furthermore, the effectiveness of the tuning strategies has been evaluated.

- A performance model for memory bound applications in multisocket systems based on an exhaustive runtime characterization on a single socket to determine the best configuration of number of threads and thread distribution among multiple sockets.
- A performance model based on memory footprint for OpenMP memory bound applications. The objective of this model is to reduce the runtime overhead by characterizing the application before the execution. This is done by profiling the spatial address pattern of the applications and a performance sampling from small workloads of the application.

1.5 Thesis outline

The work presented in this thesis is divided in the following chapters.

- **Chapter 2: Background**; In this chapter we present an overview of general High Performance Computing and a more detailed description of multicore and OpenMP environments. Following this we describe monitoring, performance analysis and tuning tools in the context of HPC with a special attention on tools used along the research of this thesis.
- **Chapter 3: Methodology for developing tuning strategies for OpenMP applications**; In this chapter, it is presented and described the methodology and the performance analysis of OpenMP applications in HPC environments with the aim of providing a dynamic tuning strategy based on a performance model for a relevant performance factor.
- **Chapter 4: Methodology application to a case study**; In this chapter, the methodology is applied to the case study of NAS parallel benchmarks. The first part of the chapter is focused on characterizing the benchmark suite in order to identify relevant performance factors, while the second part of the chapter, is dedicated to describe a tuning strategy based on an exhaustive characterization of possible configurations in of number of threads to minimize a performance factor based on memory contention for the SP benchmark.
- **Chapter 5: Performance model based on runtime characterization**; This chapter introduces a performance model based on runtime characterization on a single

socket which allows to estimate a configuration of number of threads and thread distribution to improve performance on a multicore multisocket system.

- **Chapter 6: Performance model based on profiling the memory footprint**; This chapter presents a runtime performance model based on the pre-characterization of the memory footprint of the application for small workloads and a dynamic tuning strategy to estimate a configuration of number of threads and thread distribution to improve performance by detecting and avoiding memory contention.
- **Chapter 7: Conclusions**; This chapter presents the experiences gained and conclusions derived from this thesis. It is also described the viable open lines that can be considered in the future in order to provide further strategies and performance models in the area of dynamic tuning of parallel applications.

2
Background

"Ignorance might be bliss for the ignorant, but for the rest of us it's a right fucking pain in the arse :)"

twitter @rickygervais – (comedian) **Ricky Gervais**

In this chapter we present an overview of the current state of High Performance Computing and a more detailed description of multicore and OpenMP environments. Following this we describe monitoring, performance analysis and tuning tools in the context of HPC with a special attention on tools used along the research of this thesis.

2.1 High performance computing

High Performance Computing (HPC) or supercomputing is the technology and the research field in computer engineering which aims for reducing the time required for solving computational problems, enhance its productivity, and enlarge its size and complexity. To make this possible supercomputers or clusters of computers working in parallel are used to obtain the application's solution.

Supercomputers are the hardware platforms designed to obtain the maximum performance on an application execution, but they are also research laboratories that allows users to model and simulate solutions to problems for a wide range of scientific disciplines.

In computer science, problems are defined as algorithms, which is a self-contained step-by-step set of operations to be performed for solving a specific problem. The way that algorithms are historically described is sequential, which is the easiest way for humans to perform a formal description of the problem. However, by analyzing algorithms it is possible to identify sequences of instructions that can be executed in parallel by a computational system.

The history of supercomputing starts with machines fully designed for the HPC purpose. However, nowadays they are built using commodity devices mainly to reduce costs. That is why, the evolution of commodity processors has a significant impact on supercomputers performance, in particular since the becoming of multicore processors.

Commodity multicore processors appeared on 2001 [11], and before this, market strategies tend to provide better performance by increasing the clock frequency. Multicore processors started to revolutionized the market as a solution for the lack on improvement and power consumption on frequency scaling. Initial multicore designs were focused on replicating some processor units within the same chip, but nowadays, processors integrate dozens of cores specially designed for a coordinated parallel execution (e.g. accelerators and coprocessor), which are commonly named as manycore processors. Currently, multicore and manycore processors are dominant in supercomputers.

The architecture design of current supercomputers is based on the interconnection of a massive number of compute nodes containing one or multiple processors. This design allows two scopes in parallelism utilization, the distributed memory on separate nodes and shared memory within the node. These scopes of memory can be combined in a hybrid parallelization.

Distributed memory systems in a parallel execution requires a interconnection of computing nodes, a coordination and data transfer along the execution. In HPC systems it is possible to use a vendor Message Passing Interface (MPI) implementation. MPI is a standardized and

portable message-passing library interface specification oriented to the parallel execution in distributed systems.

Shared memory systems in a parallel execution are usually supported by the operating systems through the utilization of threads. On the one hand, POSIX threads provide a standardized and portable low-level application program interface (API), which allows a extremely fine-grained control over thread management such as creation, synchronization, mutual exclusion mechanism, and so on. On the other hand, a set of different high level programming models provide an abstraction of threads being OpenMP, OmpSs, intelTBB, OpenACC, and Cilk some of the most common in HPC systems.

Cilk extends programming languages such as C and C++ with constructs to express parallel loops and the fork-join paradigm. It has been originally designed at Massachusetts Institute of Technology (MIT) Laboratory for computer Science. Intel is the current developer of an increased compatibility of Cilk with existing C and C++ named Cilk Plus. One of the most interesting features of Cilk is the work-stealing scheduling policy used to balance workload among threads or working units. In this policy, every processor maintains a stack for storing frames whose execution has been suspended, and when a processor remains in idle state, it tries to randomly steal suspended jobs from other processor's stacks.

IntelTBB is a C++ template library developed by Intel for writing programs that take advantage of multicore-processors. The library consists of data structures and algorithms to abstract the utilization of threads on the development of parallel programs. IntelTBB is an implementation based on task parallelism and uses a work-stealing task scheduling policy. Parallel template patterns hide the manipulation of tasks and they are accessed through interfaces for pipelined execution, parallel loops, parallel reduction over a range, work partitioning for parallel loops, and more.

OmpSs [12] is a programming model designed to extend the OpenMP programming model with new directives and to support asynchronous parallelism and heterogeneity. OmpSs has been developed at the Barcelona Supercomputing Center (BSC) with the aim of providing a framework for improving research and productivity. OmpSs environment is composed by a set of flexible and modular tools for compiling (Mercurium [13]), executing (Nanos runtime [12]) and analyze performance (Extrae [14] and Paraver [15]) of OmpSs applications. Its development and research has directly influenced the current OpenMP specification.

The Open Multi-Processing (OpenMP) is a high-level standardized API for developing highly-scalable and portable parallel applications. OpenMP consists on a set of compiler directives, library routines, and environmental variables to manage threads in a parallel executions.

It also provides a unified code for both serial and parallel applications: OpenMP constructs are treated as comments when the application is compiled to be executed sequentially.

The development of parallel applications in HPC requires a fundamental knowledge of parallelism and the programming model, however, obtaining the best performance usually requires a deep knowledge of the executing environment. The performance analysis is a field of study dedicated to evaluate performance, identify performance bottlenecks and provide specific solutions, however, providing general solutions is far more complicated. Through the expertise obtained in performance analysis, it is possible to define the rules to automatically identify a performance problem and apply a dedicated solution.

Performance tuning is the action performed to improve performance given a performance problem. This can be performed at the application, at hardware level or at the library responsible of managing parallelism. For the first case, the application source code is not always provided, and for the second one, making hardware modifications can be more difficult or expensive. Furthermore, these approaches provide an ad-hoc solution. To perform a more general tuning strategy it is possible to take advantage of the runtime management library. Parallel manager libraries also depend on the programming model and implementations, but they provide an interface which can be used to apply dynamic tuning strategies. With this aim, dynamic instrumentation tools such as Intel Pin, Dyninst or linker preload techniques can be used to dynamically tune the application at runtime .

Finally, because OpenMP is a standardized API and a programming model widely used in HPC environments, and with the aim of providing the most extensible solution, the analysis of performance and the development of tuning strategies for OpenMP applications can generate a great impact on HPC environments. To do this, we research on performance analysis for OpenMP applications in multicore environments to provide the identification of relevant performance problems and the definition of tuning strategies to be applied at manager library level.

Following this, we describe the most relevant aspects in our research by introducing in the following sections: Multicore architectures, OpenMP specification, and tools for characterizing, monitoring and performing dynamic instrumentation.

2.2 Multicore architectures

Multicore architectures were mainly designed to avoid three hardware design walls. First, the **instruction level parallelism** (ILP) wall because the increasingly difficulty of finding

enough parallelism in a single instruction stream to keep the processor busy, secondly, the **power wall**, when processors tend to improve performance based on increasing the frequency and consequently the power consumption, and finally, the **memory wall** because of the increasingly gap between processor and memory speeds.

The first commercial multicore processor was the 2001 IBM's POWER4 [11]. This 1GHz processor integrates more than 170 million transistors at a scale of 180 to 130nm providing two identical cores with speculative super-scalar out-of-order execution design. With a theoretical eight instructions issued per cycle and a sustained completion rate of five instructions per cycle.

Nowadays, the most common number of cores per socket in HPC systems is 8 cores (as shown in Figure 1.1), but today's Intel Xeon Phi (Knights Corner) coprocessors provide up to 61 cores [16] in a single device.

Multicore processors take benefit of the high integration capabilities to encapsulate multiple functional processors (cores) within a die. Because of the high density and integration degree, they are designed to share some modules and interfaces, for this reason, it was also required to integrate the functionality to concurrently access shared elements and maintain coherency.

The shared elements in a multicore processor are mainly the integrated shared caches and memory interfaces. Caches are used to reduce the memory gap between main memory and core speeds by taking benefit of spatial and temporal locality. As we move farther away from the core, the memory in the level below becomes slower and larger.

In order to keep memory coherency, there are different hardware solutions. On the one hand, a coherency based on snoopy bus is a faster solution but with a withdraw that does not scale well. On the other hand, directory based schemes increase latency but improve scalability. Every mechanism can implement different coherency protocols such as MSI [17] (**M**odified-**S**hared-**I**nvalid) and derivatives (MESI [18], MOESI [19], MESIF [20], and more), Firefly protocol [21], DRAGON protocol [22], and so on.

2.2.1 State of the art multicore processors

Following the previous section, to summarize state of the art processors and to illustrate the diversity of shared memory systems used in multicore processors, we describe current tendencies in multicore designs.

Commonly, multicore processor use a hierarchical topology based on different levels of shared caches used by a set of cores. This allows a limited scalability up to tens of cores.

As an instance, IBM Power8 processor [23] described in Figure 2.1 integrates 12 cores in a single processor, and each cores has support for Symmetric MultiThreading (SMT) allowing up to 8 hardware threads per core, with an aggregated parallelism of 96 hardware threads per processor. Every core has a private 64KB L1 cache and 512KB L2 cache. The L3 cache is unified and every core has 8MB local L3, but the agreggated capacity is 96MB. Furthermore, this processor includes an off-core memory cache of 128MB.

In order to combine the full potential of several multicore processors in a node in a single shared memory system, current systems provide interfaces such as Intel Quick Path Interconnect [24](Intel QPI) or HyperTransport [25] (AMD) technologies. However, this interconnection creates a non-uniform memory access environment (NUMA), where accessing remote memory is more expensive than accessing local memory. Figure 2.2 shows an example of a multicore multisocket environment where two processors contain two NUMA clusters. Every NUMA cluster has an attached memory interface and the address space is global to all clusters. The all-to-all interconnection links between clusters allow access to remote memory in 1 hop. The aggregated parallelism in a multi-socket system can easily provide tens or hundreds of available cores.

The utilization of combined multicore resources allows a high degree of computational power and it has been one of the strategies to minimize the effect of ILP wall by increasing aggregated parallelism, and the memory wall by defining a multilevel memory hierarchy. However, regarding the power wall, the efficiency of using these environments must be taken into consideration.

Power efficiency is one of the main aspects to take into account when referring to mobile devices. For this purpose, multicore processors have been specifically designed to provide the most power to the lowest power consumption. This is related to user utilization switching between high performance utilization when using computationally intensive applications and low consumption to extend battery live for normal profile of utilization.

Figure 2.3 shows ARM big.LITTLE technology [26], an heterogeneous multicore processor which integrates two different sets of cores. On the one hand, four complex out-of-order and multi-issue pipelines Cortex-A15 processors, and on the other hand, four simple in-order with 8 stage pipelines Cortex-A7. This processor allows to balance highest performance and energy efficiency.

Energy efficiency is already a primary concern in HPC systems, and it is unanimously recognized that future Exascale systems will be strongly constrained by their power consumption, and it is a research field of study in projects such as the Mont-Blanc European Project

[27]. This project explores the possibility of using mobile commodity processors to build supercomputer, such as the big.LITTLE processor.

When referring to energy efficiency in current supercomputers, the actual systems providing the best relation between performance and power efficiency are co-processors such as the General Purpose Graphic Processing Units (GPGPU) CUDA devices (NVIDIA), and OpenCL devices (AMD/ATI Radeon's), and accelerators such as Intel Xeon Phi, and others.

Figure 2.4 shows a description of the GM107 first generation Maxwell processor from NVIDIA. This coprocessor provides 640 cores distributed in 5 symmetric multiprocessors with 4 sets of 8×4 cores. The utilization of CUDA based coprocessor is associated with the use of CUDA programming model, which translate the source code to the specific Instruction Set Architecture (ISA) of the device.

The co-processors require a host processor to initiate the parallel execution. Figure 2.5 exemplifies the execution flow on a CUDA device. First, the data must be copied from the host memory to the device memory, and following this the instructions are submitted to the device which performs the computation, and finally, the data must be copied back to the host device.

Intel Xeon Phi is the technology developed by Intel to provide high performance co-processors. These co-processors are used in the same way as CUDA devices, but implementing a x86 ISA. The upcoming Knights Landing processor technology goes further, by providing a processors which can be used as a co-processor or as a host processors. Figure 2.6 shows the architecture design of the Knights Landing processor, containing 36 tiles in a mesh interconnection with 2 cores and x4 SMT per core in each tile. Tiles can be configured using 3 modes, All-to-All, Quadrant and Sub-NUMA clustering (SNC). All-to-All provides an uniform memory coherency between all the tiles (lower performance). Quadrant, where chip is divided into four virtual quadrants providing lower latency and higher bandwidth and it is software transparent. Finally, Sub NUMA Clustering, each quadrant is exposed as a separate NUMA domain to the operating system with lowest latency of all nodes but software requires NUMA optimize to get benefit.

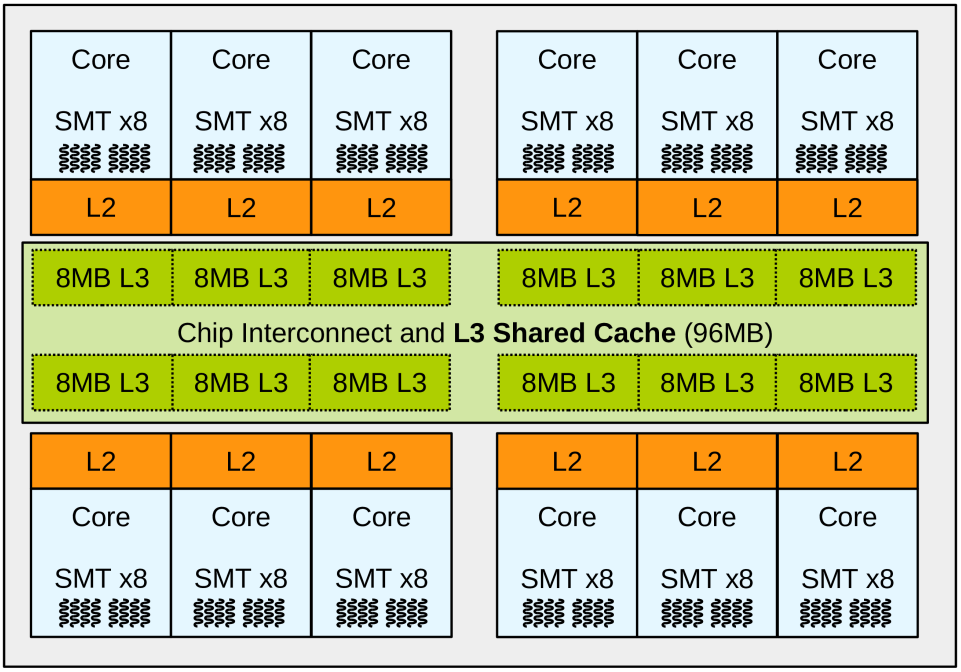


Figure 2.1: Power8 processor from IBM. A processor based on multi-level cache hierarchy of 3 levels in processors, and an external L4 cache. This processor provides SMT of 8 hardware threads per core. cores with L3 cache partitioning.

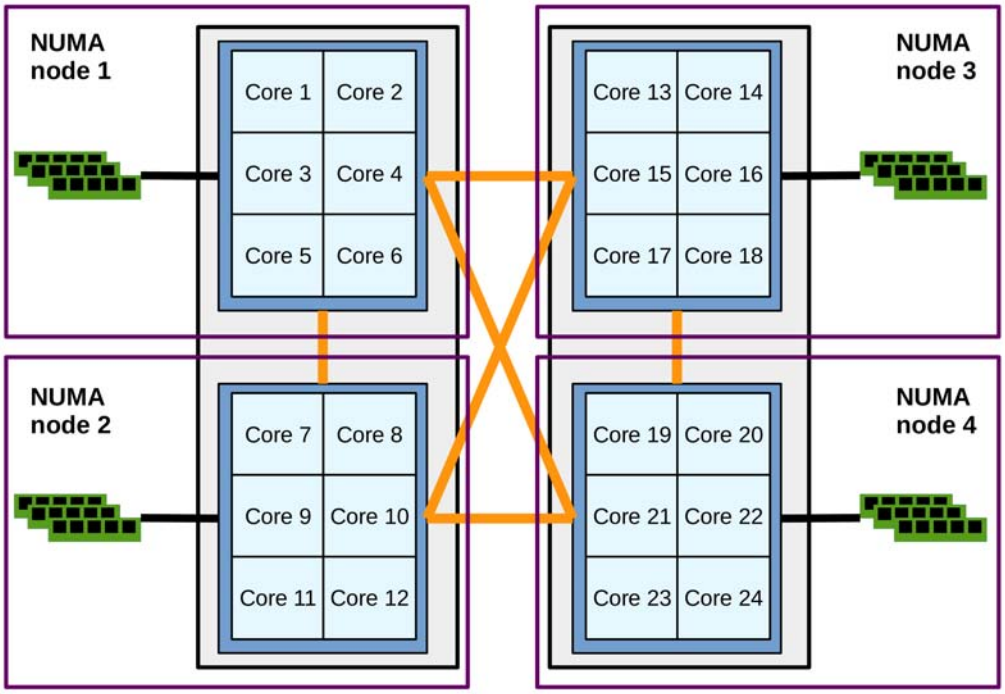


Figure 2.2: NUMA environment with 2 processors containing 2 NUMA clusters each.

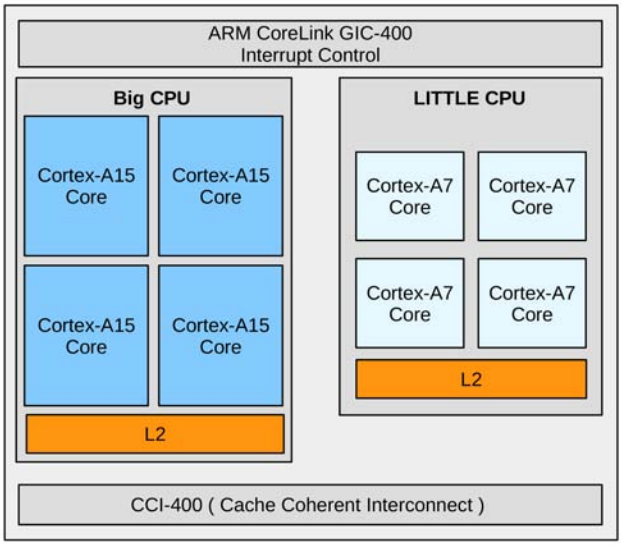


Figure 2.3: big.LITTLE processor from ARM. Heterogeneous processor with a set of 4 faster Cortex-A15 cores for high performance and a set of 4 Cortex-A7 in order cores for power efficiency.

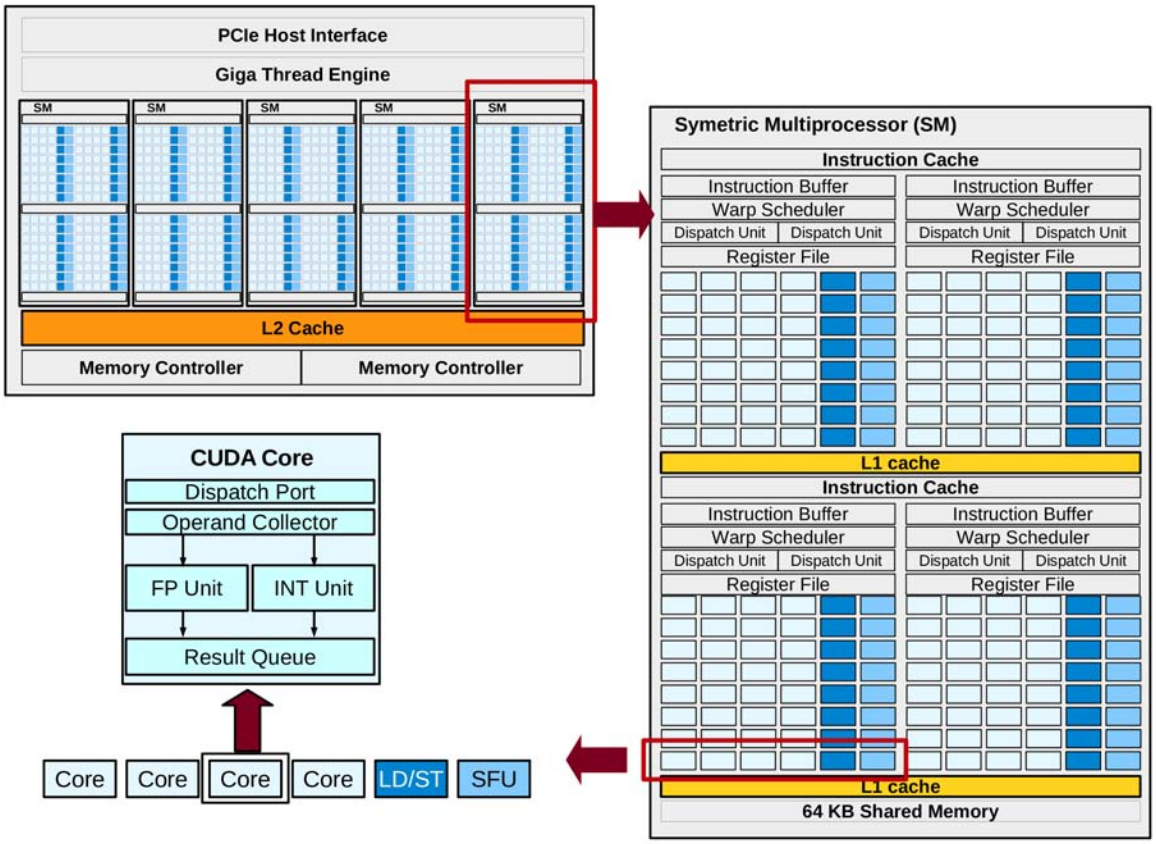


Figure 2.4: GM107 Maxwell processor from NVIDIA, a multi/many-core coprocessor device.

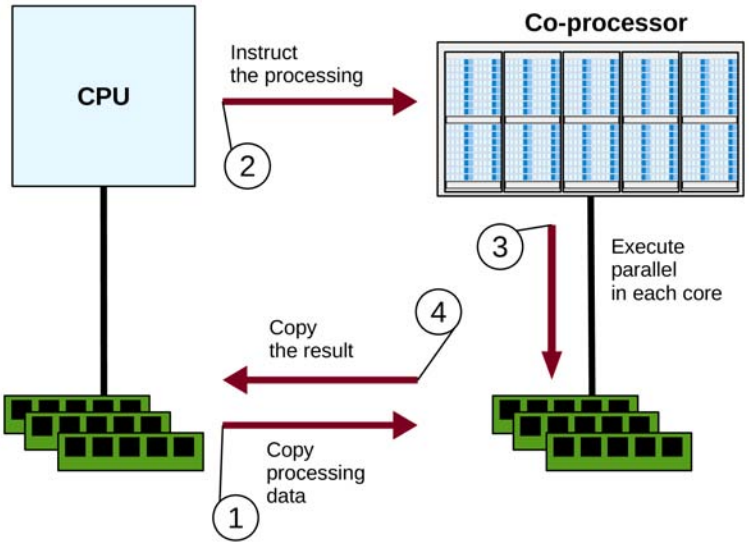


Figure 2.5: Coprocessor execution flow on a CUDA device.

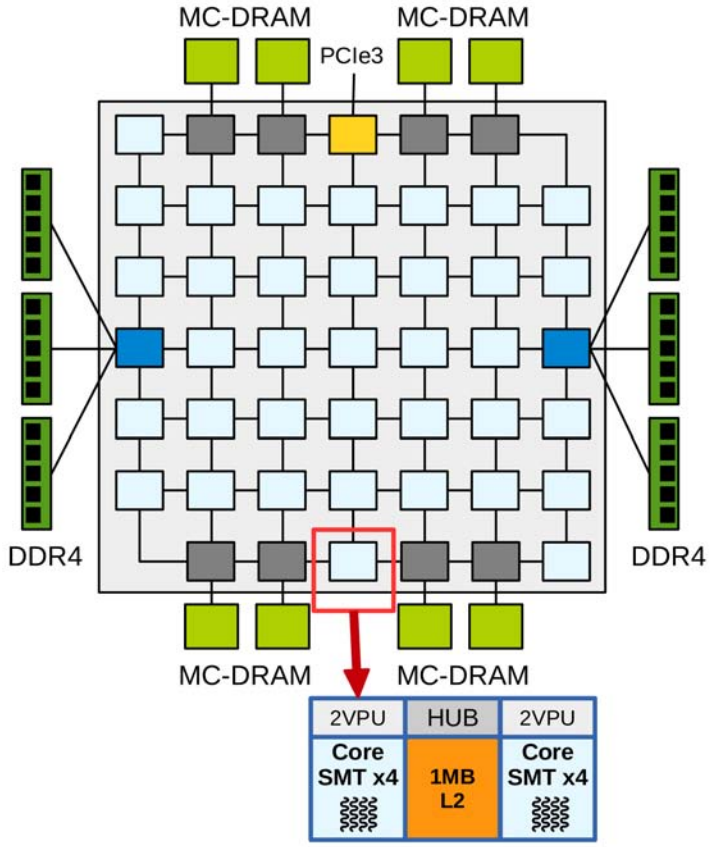


Figure 2.6: Knights Landing processor from Intel, with a 2D mesh tile interconnection of 36 tiles, were every tile contains 2 cores with SMT of 8 hardware threads per each.

2.3 OpenMP specification

OpenMP (Open Multi-Processing) [28] is an Application Program Interface (API) that supports multi-platform shared memory multiprocessing programming. It combines a set of compiler directives, library routines, and environment variables that can be used to manage parallel programs while permitting portability of C, C++ and Fortran programs.

OpenMP extends the programming language to a parallel programming model based on fork/join execution model. However, the model also considers implementation design patterns such as single-program multiple data (SPMD) constructs, loop-level parallelism, which allows the definition of parallel algorithm patterns based on data-parallelism, task-parallelism, pipelining, or geometric decomposition.

OpenMP is managed by the non-profit technology consortium OpenMP Architecture Review Board [29] (or OpenMP ARB), composed by hardware and software vendors such as AMD, ARM, Cray, Fujitsu, HP, IBM, Intel, Micron, NEC, NVIDIA, Oracle Corporation, Red Hat, Texas instruments, and more.

First OpenMP specification appeared on 1997 for Fortran programming language, and in 1998 for C/C++ programming languages. The first environments to take benefit of this API were symmetric multiprocessor architectures (SMP) such as Uniform Memory Access (UMA) architectures and Non-Uniform Memory Access (NUMA) and their variants. However, currently the environment for the execution of OpenMP programs is becoming more complex and specialized with the income of new SMP architectures such as the heterogeneous processors (e.g. ARM big.LITTLE) or the co-processors (e.g. Intel Many Integrated Cores MIC). For this reason the specification is successfully evolving.

The initial specification defines a parallel programming model mainly focused on data parallelism, were the compiler transforms parallel regions into template structures making use of OpenMP routine libraries which are going to control the runtime execution by coordinating the thread execution and define the work partitioning. These libraries usually rely on a thread interface such as the POSIX threads API to manage threads at runtime.

On the most current version of the specification, OpenMP includes task parallelism, support for accelerators, thread affinity and more capabilities providing a more flexible parallel programming model according to the evolution of multicore/manycore architectures.

Following this, we present a brief description of relevant elements within the evolution of the OpenMP specification, and a description of common parallel programming patterns in OpenMP;

OpenMP v1.0

- **Parallel control structures**; governs flow of control in the program (`parallel` directive)
- **Worksharing**; distributes work among threads (`parallel for` and `sections`)
- **Data environment**; scopes and variables (`shared` and `private` clauses).
- **Synchronization**; coordinates thread execution (`critical`, `atomic`, `barrier`).
- **Runtime functions and environmental variables** (`omp_get_num_threads`, `omp_set_schedule`, etc.).
- **Nested parallelism**; OpenMP uses a fork-join model of parallel execution. When a thread encounters a parallel construct, the thread creates a team composed of itself and some additional (possibly zero) number of threads. The encountering thread becomes the master of the new team.

OpenMP v2.0

- Added `num_threads` clause which allows a user to request a specific number of threads for a parallel construct.
- `Copyprivate` clause added. A mechanism to broadcast a value from one member of the team to the other members. The clause can only appear on the `single` directive.
- Timing routines `omp_get_wtick` and `omp_get_wtime` performing a wall clock timing.

OpenMP v3.0

- **Task parallelism** is supported by including `task`, `taskwait` constructs.
- Combine perfectly nested loops by the clause `collapse`.
- The schedule kind `auto` gives the implementation the freedom to choose any possible mapping of iterations in a loop construct to threads in a team.
- The `omp_set_schedule` and `omp_get_schedule` will change at runtime the default scheduling policy in a loop construct.
- The `omp_thread_limit` controls the maximum number of threads participating in the OpenMP program.

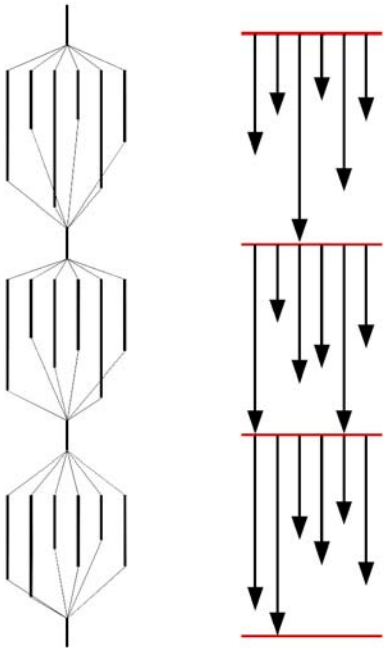
- The `omp_get_max_active_levels` and `omp_set_max_active_levels` in a nested execution manages number of nested levels.

OpenMP v4.0

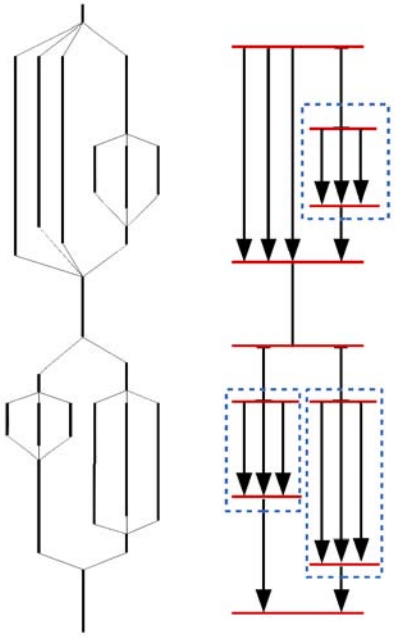
- **Support for accelerators**. The `OMP_DEFAULT_DEVICE` environmental variable, `omp_set_default_device`, `omp_get_num_devices`, `omp_get_num_teams`, `omp_is_initial_device` were added to support execution on devices.
- **SIMD constructs** to vectorize both serial as well as parallelized loops.
- **Error handling**. Capabilities to improve the resiliency and stability of OpenMP applications in the presence of system-level, runtime-level, and user-defined errors.
- **Thread affinity**. The `proc_bind` clause, the `OMP_PLACES`, and the `omp_get_proc_bind` runtime routine were added to support thread affinity policies
- Tasking extensions. deep task synchronization and task groups can be aborted. Task-to-task synchronization is now supported through the specification of `task dependency`. The depend clause was added to support **task dependencies**.
- Support for Fortran 2003. This includes interoperability of Fortran and C, which is one of the most popular features in Fortran 2003.
- **User-defined reductions**. New reduction operations `min` and `max` were added.
- Sequentially consistent atomics. A clause has been added to allow a programmer to enforce sequential consistency when a specific storage location is accessed atomically.

Parallel programming patterns on OpenMP *Scheme*

Fork-join; The fork-join model is an implementation strategy pattern which has been the basic model for the parallel execution in the OpenMP framework. In a fork-join model the main execution unit forks off some number of other execution units that then continue in parallel to accomplish some portion of the overall work. OpenMP accomplishes this by creating, executing and synchronizing threads. The OpenMP specification is flexible enough to describe or allow other low level parallel design patterns such as loop parallelism (parallel loop construct), Single Program Multiple Data (parallel construct), master/worker (master construct) or even the combination of basic design patters to create compound parallel algorithm patterns such as pipelining (sections), geometric decomposition (reductions in parallel loops).

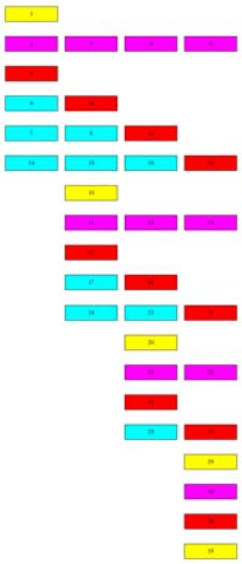


Nested parallelism; OpenMP allows for nested parallel regions during the execution. Nested parallelism can be enabled and disabled through the use of the `OMP_NESTED` environmental variable or by calling the `omp_set_nested()` routine. The support for nested parallelism expands the coverage of parallel patterns, for example, nested parallelism allows the implementation of divide and conquer patterns.

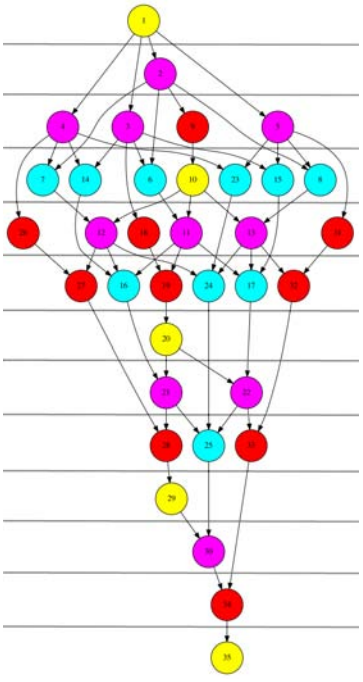


Parallel programming patterns on OpenMP *Scheme*

Task parallelism; A task parallel computation is one in which parallelism is applied by performing distinct computations or tasks at the same time. When a thread encounters a task construct, a new task is generated, but the moment of execution of the task is up to the runtime system. Execution can be either immediate or delayed. The completion of a task can be enforced through a task synchronization construct. The utilization of task parallelism tries to obtain a more efficient and more scalable utilization of resources, and also transfer more charge to the runtime scheduling.



Task parallelism with dependences; The enforced task dependence establishes a synchronization of memory accesses performed by a dependent task with respect to accesses performed by the predecessors tasks. It is responsability of the programmer to synchronize properly with respect to other concurrent accesses that occur outside of those tasks. Task dependency support involves decentralized selective synchronization operations that should scale better than taskwait-based approaches. Information about task dependencies also enables the runtime system to optimize further, such as improving task and data placement within the memory hierarchy. OpenMP defines *in*, *out*, and *inout* dependencies at the *depend* clause in the task construct. An example, the *in* dependence type defines the current task as dependent of all previously generated sibling tasks that reference at least one the list items in an *out* or *inout* dependency list. Therefore, the *out* dependence type works in the opposite way.



2.4 System characterization, monitoring, and instrumentation tools

The first rule of C.Gordon Bell on ”the eleven rules of supercomputer design” [30], is: *performance, performance, performance*.

Performance can be measured by the effectiveness of the computer system, including speed on a task completion, throughput, and/or availability. The criteria in HPC is mainly focused on task completion. Assuming this, the less time for a time completion, the best performance.

Determining the overall performance on a machine is usually done by benchmarking. For example, the TOP500 list is a ranking based on the evaluation of performance (FLoating OPeration per Seconds or FLOPS) for the execution of Linpack benchmark performing different numerical linear algebra operations.

In order to improve performance for a given application, it is necessary to monitor, analyze and tune the critical elements involved in the execution.

Processors designers are aware of the importance of providing performance information related to the hardware utilization such as cycles, instructions, cache metrics, and so on. The Performance Monitor Units (PMU) are Model Specific Registers (MSR) which can be configured to provide an insight of performance in processor utilization.

PMU registers are accessible through a kernel module and at user space through a high level Performance Application Performance Interface [31] (PAPI), *libpfm4* which provides a mapping mechanism to implementation specific hardware events (used by PAPI), or the low level *perf* which is a library that exposes the kernel performance counters subsystem to userspace code.

To evaluate the performance on parallel applications, it is necessary to identify the monitoring points and provide performance metrics. Parallel applications commonly use standardized interfaces, and tools have been defined to identify these points to obtain information along the execution. On the one hand, profilers provide a coarse grained performance information with a small overhead, and on the other hand, tracing tools provide a fine grained performance information at the cost of greater overheads.

Finally, using the collected information of an application execution, it is possible to find a candidate optimization or tuning strategy. One option is to implement an application or compiler specific optimizations at the cost of generating a non portable solution, and moreover, some problems are runtime specific only. It is possible to address runtime performance

problems by tuning the parallel library manager.

In order to maintain portability, dynamic performance instrumentation tools can be used to insert binary code to modify the parallel library manager behaviour. This is done by defining monitoring points and metrics, a performance model module to evaluate the execution and tuning strategies.

2.4.1 System characterization by benchmarking

Performance rankings in HPC hardware systems is done by benchmarking using compute intensive applications. However, it is possible to use specific applications to determine the abilities of different components of the architectures. One of the most well known bottlenecks is the memory subsystem, and therefore, memory intensive applications are used to provide empirical peaks in memory performance.

Furthermore, the utilization of a parallel paradigm via a parallel programming model can lead to performance overheads. This is going to depend on the implementation of the library. Benchmarking is also used to determine the overheads of a programming model implementation.

Following this, the benchmarks described below can be used for characterizing shared memory subsystems and OpenMP overheads.

STREAM benchmark

STREAM benchmark [32], by John D. McCalpin, is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels.

The STREAM benchmark is specifically designed to work with datasets much larger than the available cache on any given system, so that the results are (presumably) more indicative of the performance of very large vector style applications.

There are versions of STREAM benchmark using different programming models such as OpenMP, pthreads, and MPI.

It is composed of four parallel vector kernels performing the following operations:

1. Copy: $c[j] = a = [j]$
2. Scale: $b[j] = scalar * c[j]$
3. Add: $c[j] = a[j] + b[j]$

4. Triad: $a[j] = b[j] + scalar * c[j]$

STRIDE benchmark

The STRIDE benchmark [33] is designed to severely stress the memory subsystem on a node using a series of sequential kernels.

The STRIDE benchmark consists of STRID3, VECOP, CACHE, STRIDOT, and CACHEDOT. The first three benchmarks include C and Fortran language versions. All of the benchmarks utilize combinations of loops of scalar and vector operations and measure the MFLOP rate delivered as a function of the memory access patterns and length of vector utilized.

The observed rates of computation of the various access patterns within an individual test can then be compared to provide an understanding of the performance implications.

LMBench

LMBench [34] is a micro-benchmark suite designed to focus attention on the basic building blocks of many common system applications, such as databases, simulations, software development, and networking based on reported common performance problems.

The different set of tools within the suite test cached reads, memory (reads, writes, copies), context-switching, network connections, file creation and deletions, and so on.

From them, `lat_mem_rd` is used to measure memory read latency for varying memory sizes and strides. The entire memory hierarchy is measured, including onboard cache latency and size, external cache latency and size, main memory latency, and TLB miss latency. The benchmark runs as two nested loops, for the stride in the outer loop and the array size in the inner loop, and the results are reported in nanoseconds

EPCC OpenMP micro-benchmark suite

EPCC OpenMP micro-benchmark suite [35] is intended to measure the overheads of synchronisation, loop scheduling and array operations in the OpenMP runtime library.

Currently, the micro-benchmark includes an extension in order to measure the overhead of the task construct introduced in the OpenMP 3.0 standard, and associated task synchronisation constructs.

The evaluation of EPCC is used to evaluate different compilers [36] and hardware platforms [37], and it allows to expose significant differences in performance between different

OpenMP implementations or programming models.

The Barcelona OpenMP Task Suite (BOTS)

The Barcelona OpenMP Task Suite [38] (BOTS) provides a collection of applications that allows to test OpenMP tasking implementations. The different kernels allow to test different possibilities of the OpenMP task model such as scheduling alternatives, cut-offs, single/multiple generators, task tiedness, and o so on.

The Benchmark suite contains the kernels Alignment, FFT, Floorplan, Health, NQueens, Sort, SparseLU, Strassen and Unbalanced Tree Search (UTS).

2.4.2 Monitoring tools

Unix-like operating systems are the most common on HPC environments [10], and systems like Linux provide a set of generic tools that can be used to analyze performance. Pre-builded command line performance tools in Linux environments provide useful information about performance related to executed processes (`top`, `htop`), virtual memory statistics (`vmstat`), list of opened files (`lsof`), network analyzers (`netstat`), disk utilization (`iostat`, `iotop`) and so on.

Supercomputer centres require specific tools for managing distributed systems, such as the Ganglia Distributed Monitoring System [39] which is a scalable system monitor tools for high-performance computing systems .

In the context of shared memory tools such as `taskset`, *likwid* [40] and `numactl` [41], are utilities which can be used to control scheduling policies to bind processes to processors. Furthermore, *likwid* and `numatcl` are capable of characterizing the memory hierarchy. The `numactl` utility also provides a library interface to provide control for memory allocation in the shared memory hierarchy.

PAPI - Performance Application Program Interface

The Performance Application Program Interface [31] (PAPI) is a high-level API that provides the ability to start, stop and read the counters for a specified list of events in most current processors. One of the benefits of using a high-level API, rather than the low-level API, is that it is easier to use and requires less additional calls. PAPI provides a interface for C and Fortran programming models.

PAPI names a number of predefined or preset events. This set is a collection of events typically found in many CPUs that provide performance couters. A PAPI preset event name is mapped onto one or more of the countable native events on each hardware platform. Therefore, a preset event can be an available single counter, a combination of counters or unavailable.

Preset PAPI counters can be categorized into *Conditional Branching*, *Cache Request*, *Conditional Store*, *Floating Point Operations*, *Instruction Counting*, *Cache Access*, *Data Access*, and *TLB Operations*. The list of specific preset counters in a system with a PAPI installation can be obtained with `papi_avail` command.

Every different hardware platform would provide a different repertory of native events. Native events may have optional settings, or unit masks. PAPI also provides access to native events. These events can be listed with `papi_native_avail` command. As an example, in Intel *Nehalem Microarchitecture* events `L1D_CACHE_LD` counts L1 data cache read requests for the unit mask `0x0f`, but with unit mask `0x08` it provides more specific information when the cache line to be loaded is in the M (modified) state.

2.4.3 Performance analysis tools

Two types of tools can be used to develop a performance profile of an application. On the one hand, profiling aggregates statistics along the execution of the application and after the execution provides a classified and summarized report. On the other hand, tracing collects data and timestamps from triggered events. This information is called a trace and some trace formats allow their visualization. Given that the information in profiling is summarized and tracing provides information for all the events, the amount of data generated tend to be small in contraposition with traces which generates a great deal of information.

Unix-like operating systems provide a set of different tools for analyzing performance like `gprof` [42], `strace` or *sysstat* [43] utilites such as `sar`, `sadf`, `mpstat`, `iostat`, `nfsiostat`, `cifsiostat` or `pidstat`.

On HPC environments, there are different parallel oriented tools for profiling, tracing and some of them also include automatic analysis.

Common profiling tools such as *ompP* [44], *mpiP* [45], *HPCview* [46], *perfSuite* [47], *Integrated Performance Monitoring* [48] (IPM), *FPMPI-2* [49], and *Intel VTune Amplifier* [50] can be used to analyze different aspects related to performance and programming model such as OpenMP, MPI, hardware counters information, bottlenecks, and more.

Tracing tools such as *VampirTrace* [51] and *Extrae* [14][52] tools provide traces which can be visualized correspondely with *Vampir* [53] and *Paraver* [15]. Furthermore, *Open|SpeedShop*

[54] and *HPCtoolkit* [55] combine the possibility of selecting profiling and tracing analysis.

Finally, *TAU* [56] provides an application profile and automatic analysis, and *Scalasca* [57] generate automatic analysis based on traces.

We have used the following tools within the context of our research to perform the performance analysis.

OmpP – Profiling

ompP is a profiling tool for OpenMP applications. ompP's profiling report becomes available immediately after program termination in a human-readable ASCII format. ompP supports the measurement of hardware performance counters using PAPI and it supports productivity features such as overhead analysis and detection of common inefficiency situations (called performance properties).

Extrae and Paraver – Tracing and visualizing

Extrae is a package developed at the Barcelona Supercomputing Center responsible of the generation of Paraver trace-files for a post-mortem analysis. Extrae is a tool that uses different interposition mechanisms to inject probes into the target application so as to gather information regarding the application performance.

The package provides mechanism to support programming model such as MPI, OpenMP, CUDA, OpenCL, pthread, OmpSs, Java and Python. The tools is supported in Linux clusters (x86 and x86-64), BlueGene/Q, Cray, nVidia GPUs, Intel Xeon Phi, ARM and Android platforms.

Extrae gives the user the possibility to manually instrument the application and generate its own events if the previous mechanisms do not fulfil the user's needs.

Along with Extrae, Paraver is a performance analyzer based on Paraver traces with a great flexibility to explore the collected data. The combination of both tools allows the analyst to generate and validate hypothesis to investigate the trails provided by the execution trace.

2.4.4 Dynamic Instrumentation

The objective of performance analysis is to provide an insight of performance problems. This analysis can infer the elements involved in the performance problem with the aim of providing an optimization strategy.

The performance optimization can be done by modifying the application or providing a compiler optimization. However, some performance problems only arise at runtime. Tuning strategies can be performed at runtime by using dynamic instrumentation tools [58] [59] [60].

Dynamic instrumentation allows to intercept program execution in some strategic points to evaluate performance and modify the execution. Tuning strategies can be described as the combination of monitoring elements, analysis functions and tuning elements and parameters.

Following this, we describe some tools which allow dynamic instrumentation.

Intel Pin

Pin [8] is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures that enables the creation of dynamic program analysis tools.

The tools created using Pin are called Pintools, and they can be used to perform program analysis on user space applications in Linux and Windows. Instrumentation is performed at runtime on the compiled binary files.

Pin provides a rich API that abstracts away the underlying instruction-set idiosyncrasies and allows context information, such as register contents, to be passed to the injected code as parameters.

Dyninst

The *Dyninst* [9] API library provides an interface for instrumenting and working with binaries and processes.

In the framework of Dyninst, the target process to be tuned is defined as a mutatee, and the user defined tuning strategy is defined as a mutator, which uses the Dyninst API library.

There are two primary abstractions in the API, the points and the snippets. A point is a location in a program where instrumentation can be inserted, and a snippet is a representation of some executable code to be inserted into a program at a point.

Linker preload

This technique is based on function interpositions. This can be done by injecting a shared library into an application before the application gets actually loaded. If the library that is being preloaded provides the same symbols as those contained in shared libraries of the application, such symbols can be wrapped in order to inject code in these calls.

This technique is commonly used in Linux systems, and accessible by using the LD_PRELOAD environment variable.

The function interposition makes possible to modify the execution on the interposition points and also to bypass the execution to the real funciton. Performance analysis tools use this technique to capture the specification defined functions in programming models to profile and trace the applications, such as, for example, ompP profiler and Extrae tracing tools.

2.5 Related work

In the context of our research we consider different fields of study, such as dynamic instrumentation which is used in performance analysis environments to tune applications at runtime, dynamic schedulers, which interact with the runtime manager to determine the best way to execute parallel units of work, and finally performance models.

MATE[61] performs automatic dynamic tuning by inserting code into the application through the Dyninst library. This framework uses externally provided strategies for taking tuning decisions.

Active Harmony [62] is an automated performance tuning infrastructure for distributed systems, optimizing programs regarding network and systems resources. Active Harmony employs an empirical off-line auto-tuner that improves the performances of all exchangeable libraries in the program.

Autopilot [63] is another tuning environment for parallel and wide area distributed systems based on fuzzy logic to automate the decision-making process. In this case, the developer must insert sensors and actuators into the source code prior to the execution of the parallel execution.

The previous tools are focused on distributed systems but, Wicaksono et al.[64] demonstrates the functionality of a collector-based dynamic optimization framework called DARWIN that uses collected performance data as feedback to affect the behaviour of the program through the OpenMP runtime. It is able to take different actions, such as modifying the number of threads, adjusting core frequency, or memory allocations on ccNUMA systems. This environment has been evaluated on ccNUMA systems for the NAS Parallel Benchmarks Suite [65] of kernel applications (up to class B workloads). This works relies on off-line analysis of empirical collected data, and the decision making strategy requires a training for every different system, and on the other hand, our approach proposes the definition of performance models describing the relations of the elements involved in performance problems to provide

a portable solution.

Regarding scheduling strategies, works such as Stephen Olivier et al.[66] and A.Duran et al.[67], are focused on performance issues related to task managing. The former discusses task managing work-stealing algorithm, where it is necessary to define the appropriate number of tasks to be stolen from a remote queue, as well as the performance results achieved on several architectures. While the latter proposes a cut-off strategy for limiting the space required to deploy tasks using a threshold for limiting the task graph. Various strategies are evaluated to reduce the number of created tasks, and consequently the creation overhead is reduced.

In addition to task scheduling implementations, the utilization of different compilers could be determinant for tasks based applications performance, as shown in an evaluation for different compilers supporting task parallelization in Stephen Olivier and Jan Prins[68] for the evaluation of the Unbalance Tree Search (UTS) algorithm using different compilers.

Finally, the performance model Roofline [69] is designed to assist in software development and optimization by providing detailed and accurate information about machine characteristics. The Roofline model is a visually intuitive performance model used to bound the performance of various numerical methods and operations running on multicore, manycore, or accelerator processor architectures. However, the Roofline model is used to identify what is deficient but does not define how to fix it.

2.6 Summary

In this chapter, we have introduced the current state of High Performance Computing and supercomputers as its facilities. We have defined performance as the key objective on this context, and how performance has been affected by the evolution of current processors.

Parallelism is the key for improving performance and, to take benefit of parallel systems, programming models have been designed to facilitate the development of applications.

OpenMP is a standardized parallel programming model that facilitates the utilization of parallel resources in shared memory systems, such as multicore processor based systems. It is one of the most utilized programming models in HPC systems because its annotations style for expressing parallelism let developers to focus on the functional part of the development but obtaining great scalability.

Multicore architectures are inherently parallel machines that have been developed to overcome former performance issues, however, taking performance to the limit, new performance problems have arisen. Sometimes, the performance obtained do not always meet the

expectations, and performance analysis is required to identify the performance bottlenecks in order to tune the execution environment.

Performance analysis is a field of study focused on obtaining the maximum benefit of parallel systems. To do that, performance analyst require tools for characterize, monitor, and instrument applications. These tools are used the tune the execution environment to get its full potential.

Finally, we have introduced some related work on dynamic tuning and runtime scheduling. These two fields of study are related to the topic of this thesis in the sense that we provide dynamic tuning strategies to configure the runtime scheduling library. This is done in order to minimize the effect of performance degradation in OpenMP memory bound applications on multicore multisocket systems. To do that, we have developed a methodology that lead us to identify a relevant performance factors, which has been modeled to provide two dynamic tuning strategies.

3

Methodology for developing tuning strategies for OpenMP applications

"Wer mit Ungeheuern kämpft, mag zusehn, dass er nicht dabei zum Ungeheuer wird. Und wenn du lange in einen Abgrund blickst, blickt der Abgrund auch in dich hinein."

Jenseits von Gut und Böse – **Friedrich Nietzsche**

In this chapter, we present the methodology and the description of performance analysis with the aim of providing a dynamic tuning strategy based on a performance model for a relevant performance problem.

3.1 Objective

In order to obtain the maximum performance for the execution of an application in current HPC systems, it is necessary to consider the diversity of parallel contexts (distributed or shared memory) and the heterogeneity of environments (multicore processors, multisocket nodes, co-processors, etc) as factors for performance analysis.

A common consideration within the development of a parallel application is deciding whether describing a distributed memory model expecting unlimited scalability but increased overheads due to message passing protocol, or a shared memory model with memory limitation expecting no overheads but contention due to the concurrence access on shared components.

However, these initial performance assumptions can be proven wrong at runtime, and the developer would require a performance analysis and tuning of his application. This optimization in some cases requires a good understanding of the computer architecture, and automatic performance tuning tools integrate optimization solutions with the aim to extend their utilization to all kinds of users.

The objective of automatic performance tuning tools is to effectively hide the user the intrinsics of common performance problems. To do that, the performance tuning tools require the knowledge of relevant performance problems and the tuning strategies that can be applied.

As nowadays, the evolution of current HPC environments is highly related to multicore systems, is it desirable to evaluate the performance of applications on such systems, and evaluate impact of common performance problems, and new ones as well.

On the basis that multicore systems are systems designed as shared memory systems and OpenMP is the most used API for the shared memory programming model in HPC systems, and with the aim of modeling performance and providing tuning strategies, we define our specific objective as follows:

Definition of a work methodology to identify performance problems on OpenMP applications by evaluating its impact on performance, analysing relevant observable runtime metrics, and describing possible tuning strategies to be performed at runtime.

Following this, in section 3.2 we present the methodology developed with the aim of systematically developing performance runtime tuning optimization strategies for specific application patterns taking into consideration hardware characteristics. These performance optimizations are intended to be applied by means of the management code provided by most high level libraries. Finally, in section 3.3 we summarize the principal stages of the methodology.

3.2 Methodology

The objective of the proposed methodology is to guide the performance analysis needed to identify relevant performance problems and the study required to provide an analytical performance model, which, can determine parameters to tune at runtime manager library level to effectively improve the performance at runtime.

Fig. 3.1 shows a diagram of the different stages defined in order to define the tuning strategies for tuning applications based on performance models.

The diagram is composed of the main phases for the **context** definition and the problem **evaluation**. The objective of the context phases is to provide a candidate performance factor to be modeled on the evaluation phase, which, needs to define whether a tuning strategy can be applied to the manager library or not. Finally, the tuning strategy and overheads generated must be evaluated, in order to provide an effective strategy.

Firstly, it is required to identify a context for a performance problem. We assume that a performance problem is going to be expressed regarding the context. For example, an execution on a specific architecture can present a performance problem, and on a different architecture the performance problem is not detected. However, in both cases there is a latent performance factor within the context that depends on the hardware architecture to express the performance problem.

To provide a performance factor within a context, we are going to consider representative applications and performance analysis tools of HPC environments, moreover, benchmarking applications and vendor provided system information is used to characterize the systems.

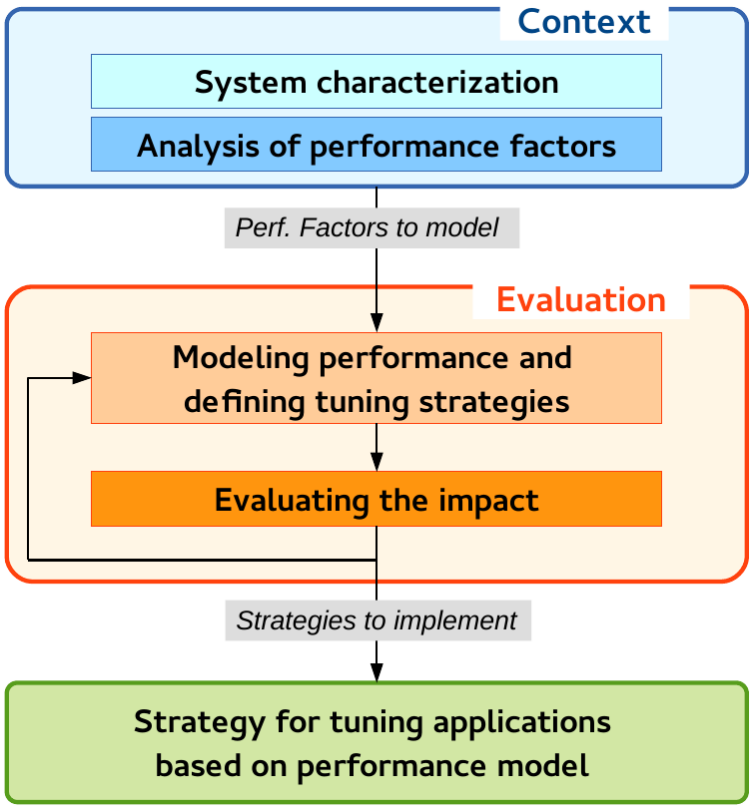


Figure 3.1: Methodology for generating dynamic tuning strategies in multicore systems

3.2.1 System characterization

In order to identify the impact of different performance problems in OpenMP applications executed in HPC multicore systems, it is necessary to describe the scope of the context to be analyzed. We propose a characterization of the application, the manager library, and the hardware. To do that, it is possible to uses different tools and benchmarks characterizing such elements. This characterization is going to provide empirical peaks and boundaries in performance.

Application

In order to identify the key elements of performance problems, the characteristic of the application must be considered, at global and at specific level.

Scientific programs can be classified at a global scope by its pattern design (Master/Worker, Pipeline, SPMD), and some specific strategies can be applied at this level of abstraction. However, common but less application dependant characteristics can be considered such

as iterative or recursive execution patterns. Furthermore, taking into consideration the performance of the parallel part of the execution, the application can be characterized as computational bounded or memory bounded.

Every parallel application is developed with a functional purpose, however, parallel applications are usually developed on the principles of structured modularity and reutilization. Parallel applications tend to utilize parallel structural patterns. The last is also emphasized by the use of standardized parallel programming models. Following this, we assume that analyzing performance and providing a performance optimization for a representative set of applications in a context will provide a solution that can be extended on applications with the same characteristics.

A representative set of HPC applications are the NASA Parallel Benchmark suite [65] (NAS benchmarks or NPB), which is composed of a set of pseudo-applications and kernels used in a wide range of scientific applications. Assuming this, a performance improvement on these functions and kernels would improve applications using them or applications with the same structural characteristics.

The NAS benchmark suite is provided with 10 OpenMP benchmarks: Five kernels:

- IS - Integer Sort, random memory access
- EP - Embarrassingly Parallel
- CG - Conjugate Gradient, irregular memory access and communication
- MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive
- FT - discrete 3D fast Fourier Transform, all-to-all communication

Three pseudo applications;

- BT - Block Tri-diagonal solver
- SP - Scalar Penta-diagonal solver
- LU - Lower-Upper Gauss-Seidel solver

Benchmarks for unstructured computation, parallel I/O, and data movement.

- DC - Data Cube

- UA - Unstructured Adaptive mesh, dynamic and irregular memory access

Application characterization can be done using profilers and tracing tools in order to determine performance problems on different parts of the application. The ompP [44] profiler provides a summarized information for all the OpenMP parallel regions, such as execution time per parallel execution element, performance counters information, and library overheads. The Extrae tracing library provides a wrapper for OpenMP applications that generates a trace of events for all parallel OpenMP constructions, which can also be combined with performance hardware counters information.

Manager library

Compiler distributions supporting OpenMP implement the specification OpenMP through a set of compiler directives and providing a dynamic library to manage the runtime API function calls. Runtime libraries examples are *libiomp* for Intel Compiler and *libgomp* for GNU GCC.

The analysis of the performance of the runtime library can be used to evaluate the library overheads and to compare different implementations of the library. The EPCC [36] OpenMP micro-benchmark suite provides overheads for synchronisation, loop scheduling and array operations in the OpenMP runtime library.

In Performance analysis of OpenMP data parallel applications the overheads of the manager library are mostly considered as a constant, however, in task parallel applications the runtime management is more relevant and consequently has more impact in performance overhead. That is, management in task parallel applications is not only generated by task constructs, but also due to scheduling infrastructure (distributed or centralized), the number of tasks and size of task queues, the task creation scheduling pattern (e.g.: breath-first, width first), time consumed on dependences analysis, and so on. A good option to measure the impact of the runtime library for an applications is to use tracing and profiling tools, for example ompP and Extrae.

Hardware

Hardware characterization can be done by considering vendor provided system information, system aware tools, and benchmarking applications.

Static architecture information is usually accessible through the operating system. This information can be summarized or enhanced by third-party utilities such as `numatcl` utility or *likwid* tool. These utilities provide a human readable memory subsystem description from the

command line. Using these tools we obtain information about number of processors, number of cores, processor frequency, memory hierarchy and memory sizes, and core associativity.

In some cases, vendor provided information defines the theoretical performance for its architecture but, in order to validate sustained peak performance, it is necessary to use benchmarking.

The objective of benchmarking is to express some intensive patterns to evaluate a specific performance metric. To evaluate compute intensiveness, Linpack benchmark can be used to determine FLOPS. To evaluate performance of memory subsystem, STREAM and STRIDE benchmarks provide the memory bandwidth evaluated on a parallel execution. Furthermore, LMBench suite provides a set of different benchmarks to measure context-switching, network connections, and also, characterize effective cache memory latencies (*lat_mem_rd*).

3.2.2 Analysis of performance factors

Presumably, the strategies applied for solving a problem within an application could be applied for other applications in the same context. By assuming this, for a given performance problem, we propose to develop a performance model to provide an early detection based on runtime monitoring to determine the best configuration of the environment.

The objective of the analysis of performance factors, is to identify latent performance problems to model. A performance factor is a latent performance problem only expressed under certain conditions, for example, depending on the workload and degree of parallelism. The performance problem has to significantly impact on performance and represent a common pattern in HPC applications.

An analysis of performance factors requires to identify problems. To do that, it is necessary to describe where in the parallel execution is the problem located, which is the performance factor, and what are the consequences of the realization of the factor on the application performance. This must be done considering the application characteristics, the runtime manager library or the hardware architecture.

In figure 3.2 we describe some common considerations to be taken on performance analysis for OpenMP applications executed in multicore environments.

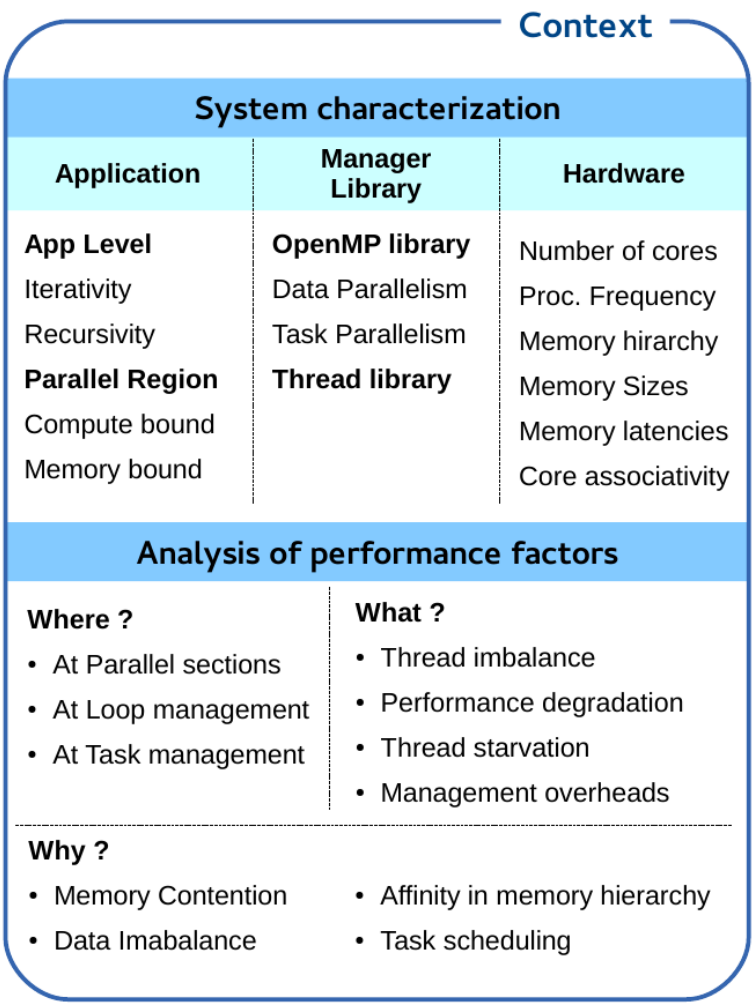
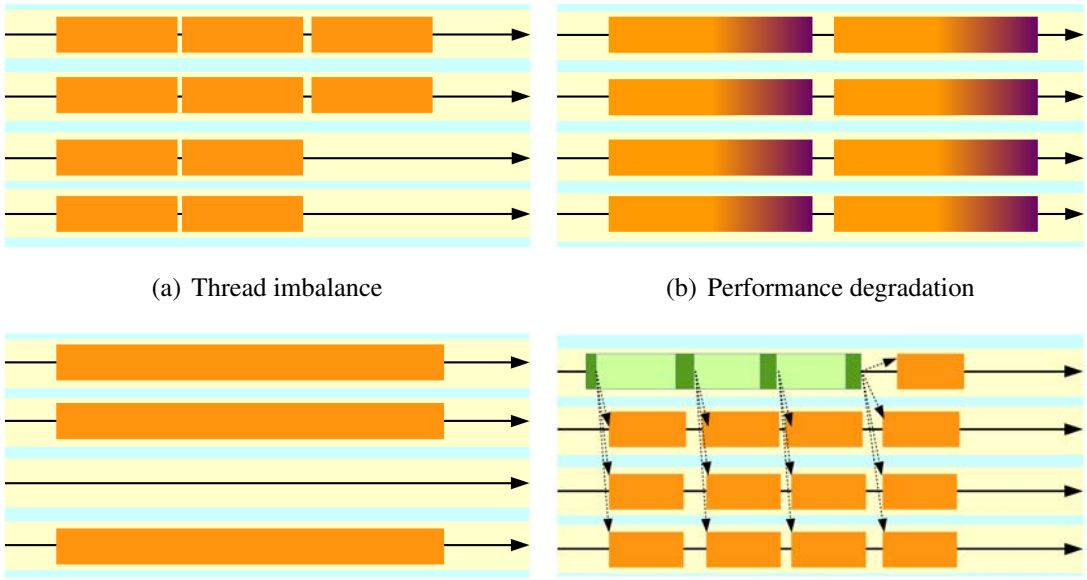


Figure 3.2: Context characteristics development template

Following this, in the context of this research, we describe some recurrent performance problems from the literature [70], [71], [72], [73] to consider under the performance factor analysis.

- **Thread imbalance**; the execution time between threads along the execution of a parallel region is significantly different.
- **Performance degradation**; the execution time at thread level given a fixed workload is significantly different when changing the degree of parallelism and threads distribution policy. When compared, it can be observed that the execution time per workload unit is increased.

- **Thread starvation**; some of the threads remain idle along the parallel execution for a given workload and degree of parallelism.
- **Management overheads**; the overhead required for managing the parallel execution dominates performance compared to the computational part.



(a) Thread imbalance (b) Performance degradation

(c) Thread starvation (d) Management overheads

Figure 3.3: Performance problems on OpenMP applications

The previous problems can be caused by one or a combination of the following performance factors.

1. **Memory contention**; data dependencies among threads or concurrent threads accessing a shared memory interface can cause memory contention generating imbalance or performance degradation. On the one hard, data dependencies can be inherent to the data access pattern and, consequently, prefetching strategies or cache line data alignments can alleviate the performance problem. On the other hand, cache contention can be produced by the limitation of bandwidth, high latencies or small sizes in shared memory devices. In some cases, it can be useful to balance the performance degradation and the degree of parallelism to minimize the effect of performance degradation.

2. **Data imbalance**; differences in the amount of data assigned to each thread can cause thread imbalance. This performance factor can be implicit to the application workload. However, adequate dynamic strategies can be applied for data partitioning and scheduling such as data reordering.

3. **Affinity in memory hierarchy**; when an application is executed on an architecture, a mismatch between the application configuration and computing elements sharing memory resources can originate performance degradation. Trying to take advantage of collaborating threads and memory re-utilization sometimes can cause a false-sharing condition. Analyzing the behaviour of the application and assigning collaborative threads to the appropriate hierarchy levels will promote collaborative computation.

4. **Task management** [66],[67], [74]; there is an extensive variety of performance factors in the task parallel model. Tasks, as executing units, are also affected by the same performance factors described above. Furthermore, due to its highly dynamic nature, they also present specific performance factors such as those originated at application level due to task creation patterns or task dependencies; as well as at library level due to task creation overhead, task scheduling policies, the amount of tasks created, and memory utilization per task. Some strategies can be applied, such as adapting dynamically the granularity of the task level parallelism, and adjusting the scheduler behaviour to be aware of the application and hardware requirements.

Because some performance problems only occur depending on the context, in order to characterize them, it is necessary to evaluate different configurations of the application, and execute on different environments.

A first approach is to perform a *strong scalability* evaluation by fixing a workload for the application and evaluate the performance obtained partitioning the problem using different number of threads.

A *week scalability* evaluation is based on fixing a workload per computational unit to evaluate linear scalability. To accomplish that, the execution time should stay constant while the workload is increased in direct proportion to the number of processors. This scalability analysis is widely used in distributed systems to evaluate nearest-neighbour communication patterns. In shared memory systems, this analysis can be used to determine scalability on NUMA systems because using more processors would increase the per processor memory and data locality.

The previous analysis can be based only on the execution time, but in order to obtain a fine grain analysis it is possible to use profiling and tracing tools providing summarized information at different levels.

To refine the analysis, the information provided by hardware counters can be used to evaluate the behaviour of the hardware architecture, in order to identify limitations or bottlenecks.

3.2.3 Modeling performance and defining tuning strategies

Performance modeling can consist in representing performance with analytic expressions.

In parallel executions the optimal performance is accomplished when the parallel execution time ($T(n)$) is proportional to the number of computational resources (n);

$$T(n) = \frac{T(1)}{n} \tag{3.1}$$

However, this assumption can only be done in the parallel parts of the application, as pointed by the Amdahl's law. The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.

$$\begin{aligned} & n \in \mathbb{N} \\ & B \in [0,1] \\ & T(n) = T(1)\left(B + \frac{1}{n}(1-B)\right) \end{aligned} \tag{3.2}$$

Therefore, the theoretical Speedup(n) that can be obtained by executing a given algorithm on a system providing n resources is:

$$Speedup(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1-B)\right)} = \frac{1}{B + \frac{1}{n}(1-B)} \tag{3.3}$$

In computer architectures the Speedup metric is used to describe the relative performance improvement when executing a task. Moreover, this metric can also be used when evaluating a tuning strategy in order to compare the default execution environment against the tuned execution.

When running an algorithm with linear Speedup, doubling the number of processors

doubles the speed. The Efficiency is a value, typically between zero and one, estimating how well-utilized the processors are in solving the problem, compared to how much effort is wasted in communication and synchronization. Efficiency is defined as follows:

$$E(n) = \frac{Speedup(n)}{n} = \frac{T(1)}{nT(n)} \tag{3.4}$$

These are fundamental expressions to model performance. However, in this thesis proposal, analytic expressions must represent and consider metrics from the hardware architecture, runtime configuration and application in order to allow the prediction of the application performance for different execution contexts and provide configurable parameters for the tuning strategy.

Finally, considering a model providing the tuning parameters, it is necessary to define the tuning strategy to apply at runtime. To do this, dynamic instrumentation tools such as Intel Pin, Dyninst or Linker preload techniques can be used to create a interposition function layer in order to interact with the implementation of the programming model to integrate the monitoring, analysis and tuning of the runtime library.

3.2.4 Evaluating the impact

The utilization of a interposition library containing the required instrumentation, analysis and tuning must consider the overheads derived from the tuning tool and the tuning strategy. The evaluation of the overheads measures and compares an execution with no instrumentation against the tuned execution.

Furthermore, the impact evaluation requires to estimate the quality of the performance model predictions. Therefore, the evaluation phase is iterative because this quality refinement could require a redefinition of the model or the tuning strategy.

3.3 Summary

In this chapter, we have described a methodology to define tuning strategies for OpenMP parallel applications based on the definition of performance models. To define a performance model it is required to accurately describe a picture of the execution context to be modeled and the relation of the key performance elements.

Firstly, the methodology describes a **context** stage to identify a relevant performance factor to be modeled. This is done by characterizing a set of relevant applications in the context of HPC.

Because the degree of complexity required on defining a performance model, we define an **evaluation** iterative stage to allow the redefinition of the performance model and the tuning strategy. This decision has to be taken by evaluating the impact of the proposed solution.

Finally, to exemplify the proposed methodology, an experimental evaluation of the model is shown in the following chapters of this thesis.

4

Methodology application to a case study

"Cuando hayas acabado no habrás hecho más que empezar."

Todos ellos – (songwriter) **Nacho Vegas**

In this chapter, the methodology is applied to the case study of NAS parallel benchmarks. The first part of the chapter is focused on characterizing the benchmark suite in order to identify relevant performance factors, while the second part of the chapter, is dedicated to describe a tuning strategy based on an exhaustive characterization of possible configurations in of number of threads to minimize a performance factor based on memory contention for the SP benchmark.

4.1 Methodology application on NAS Parallel Benchmarks

In order to exemplify the use of the proposed methodology, in this chapter we show its application on the NAS Parallel Benchmark suite, which is a representative set of kernels and pseudo applications used in scientific applications.

Section 4.2 presents an analysis to identify a relevant performance factor within the set of benchmarks on two different system architectures.

Section 4.3 introduces a tuning strategy and its evaluation developed to minimize performance degradation by dynamically tuning the number of threads at parallel region level. This is done by taking benefit of the iterative pattern of the application. This section replicates the previously observed performance problem on two new hardware systems to evaluate the impact of the performance problem in different architectures.

4.2 Context analysis for the identification of performance factors

The NAS parallel benchmarks version evaluated is the 3.3.1, which provides benchmarks for the programming models MPI, OpenMP and a hybrid versions of MPI+OpenMP. The OpenMP distribution consist of 10 different OpenMP benchmarks (C and Fortran); all of them implement a data parallel paradigm.

In order to execute and validate its execution, each benchmark provides a set of predefined workloads named Classes and labelled S for small, W for workstation, and a bigger set of workloads starting from A to E. Moreover, every predefined executed workload is validated by comparing the execution results against the pre-calculated solutions within the benchmark.

4.2.1 System Characterization

Hardware characterization

FatNode; The evaluation environment for the NAS benchmarks has been developed in a FatNode of the SuperMUC supercomputer at the Leibniz-Rechenzentrum (LRZ). The node is composed of four Westmere-EX Intel Xeon E7-4870 processors at 2,4GHz, and an aggregated 64GB main memory per processor, which provides a total of 256GB of shared memory.

Figure 4.1 shows the characteristics of the processor. It is composed of 10 cores with Hyperthreading capabilities.

Each processor has a separated data and instructions level 1 cache of 32KB, and a private level 2 cache of 256KB. This version of Westmere-EX provides 30MB of unified level 3 cache or LLC. However, this LLC is partitioned, meaning that there is a slice of the L3 for each core o group of cores, and data is shared along a ring interconnection.

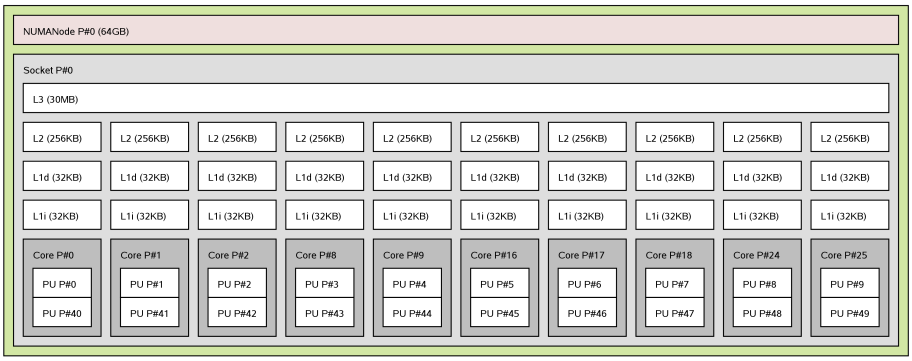


Figure 4.1: Processor architecture on System FatNode

Figure 4.2 shows the execution results of the benchmark `lat_mem_rd` in order to characterize memory latencies within the different levels of the memory subsystem. This benchmark measures memory read latency for varying memory sizes and strides. The results are reported in nanoseconds per load. The entire memory hierarchy is measured, including onboard cache latency and size, external cache latency and size, main memory latency, and TLB miss latency. However, current architectures provide latency hiding techniques such as prefetching, cache coherency, and relaxing the memory consistency requirements.

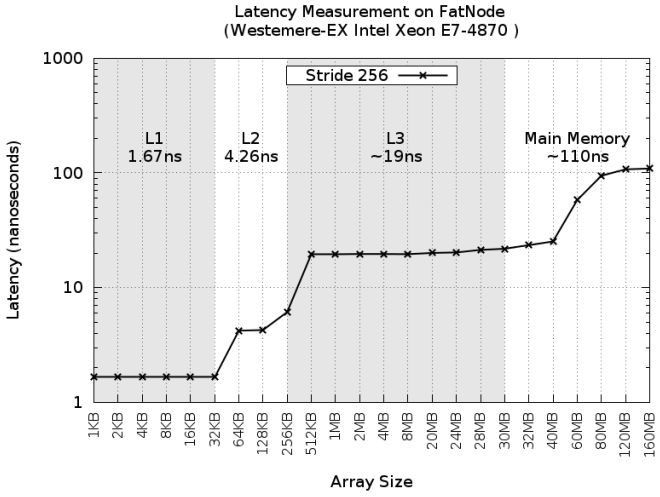


Figure 4.2: Evaluation of memory latencies on FatNode system with `lat_mem_rd`

System T7500; The evaluation of the performance factor based on performance degradation is evaluted on the T7500 system provided by the CAOS department at the UAB. This system is composed of two Intel Xeon E5645 processors developed with the Westmere micro-architecture. Figure 4.3 depicts the processor architecture. It can be observed that it is composed of two sets of three cores encapsulated in a single die. It provides an unified L3 cache of 12MB (6MB+6MB). Each processing element is a x2 multithreaded core with separated 32KB L1 cache for instruction and data, and individual L2 of 256KB.

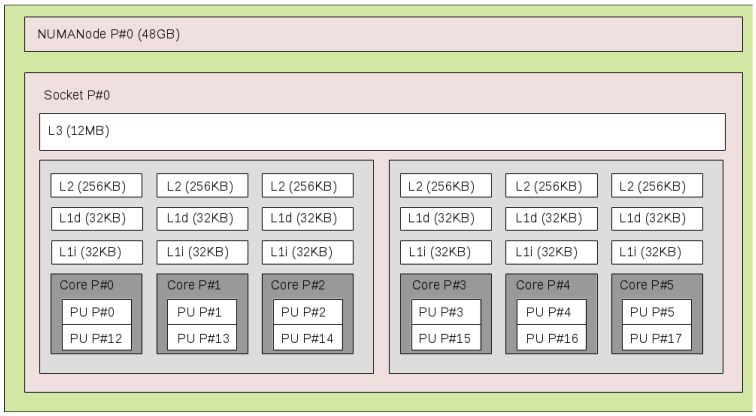


Figure 4.3: Processor architecture on System T7500

Figure 4.4 shows the execution results of the benchmark `lat_mem_rd`. The experimental results shows the evaluation for different strides in order to identify the saturated latency values corresponding to the different levels of caches. Level 1 cache has 1.5ns latency, Level 2 cache 7.73ns latency, Level 3 cache about 19ns, and local access to main memory requires around 55ns.

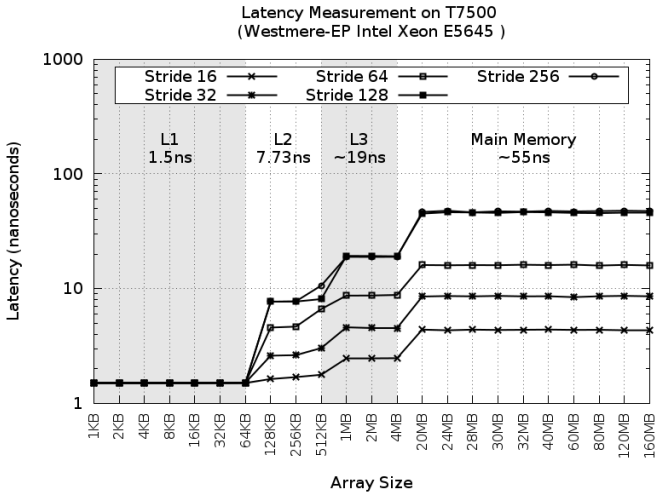


Figure 4.4: Evaluation of memory latencies on t7500 system with `lat_mem_rd`

Manager library characterization

In order to characterize the application using the NAS parallel benchmarks, firstly, an analysis of scalability for each different benchmark and different workloads is presented. The results have been obtained with the ompP profiling tool and, for each benchmark, figures for the execution time, efficiency and overhead analysis are provided.

The execution time and efficiency show the execution time using a fixed workload and different number of threads. Ideally, the best scalability will follow the ideal scalability equation, Eq.3.1.

We use the method described by Fürlinger and Gerndt in [70] to analyze the overheads limiting the scalability of OpenMP applications in order to identify performance problems in our experimental context. To do that, we use the ompP profiler report information showing a cumulative per thread analysis of time consumption between different parts of the management library. In this representation the ideal execution is defined as a constant cumulative time (black ruler). The different overheads are defined as follows:

- **Management**; startup time or thread creation, and shutdown or thread destruction in `parallel`, `parallel_loop`, `parallel_sections` and `parallel_workshare` OpenMP constructions.

- **Limited parallelism**; time consumed in the implicit exit barrier for `sections` and `parallel_sections` OpenMP constructions.

- **Imbalance**; time consumed in the implicit exit barrier for `loop`, `workshare`, `single`, `parallel`, `parallel_loop`, and `parallel_workshare` OpenMP constructions
- **Synchronization**; execution time consumed on accessing or consimung locked regions such as `atomic`, `barrier`, `flush`, `critical`, and `omp_set_lock` constructs.

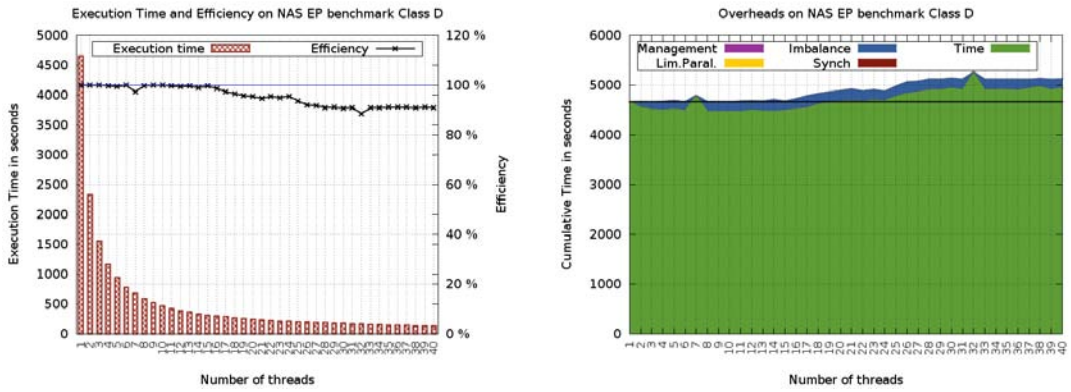
Application characterization

In this evaluation, a characterization of the NAS Parallel Benchmarks using different workloads is presented by performing exhaustive executions using different number of threads.

The execution environment is a FatNode of the SuperMUC supercomputer, which provides up to 4 processors, and each processor contains 10 cores. In these experiment Hyperthreading capabilities have not been evaluated. Therefore, the maximum degree of parallelism in this context is a 40 threads execution.

- **EP (Embarrassingly parallel)**; This kernel generates pairs of Gaussian random deviates according to a specific scheme. The goal is to establish the reference point for peak performance of a given platform.

  Figure 4.5(a) shows that the scalability is this system is high, and the efficiency greater than 80%. Regarding the per thread overheads, there cause seems to be a small increase in thread execution time and imbalance. No significant performance problems or factors can be observed.



(a) Exection time an efficiency on EP.D (b) Overhead analysis on EP.D

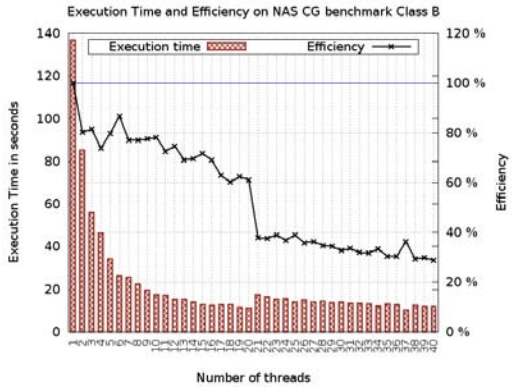
Figure 4.5: Scalability analysis on EP benchmark

- **CG (Conjugate Gradient)**; CG uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix.

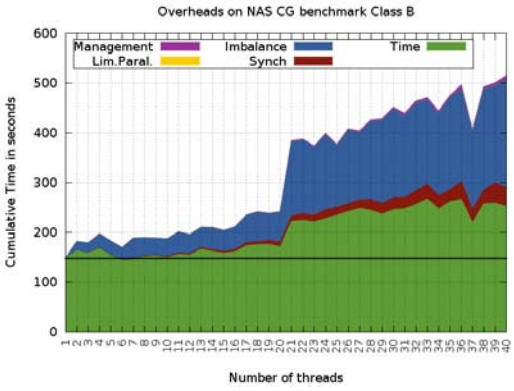
  Figure 4.6(a) shows that the overall performance does not scale for more than 20 threads and a performance degradation when using 20 to 21 threads, which is probably due to the latency increment on the third socket accesses. Figure 4.6(b) shows the relevance of synchronization elements, which at the end, can lead to thread imbalance.

  However, Figures 4.6(c) and 4.6(d) shows a performance evaluation for class D with completely different results. Efficiency is maintained over 80%, and significant overheads are observed.

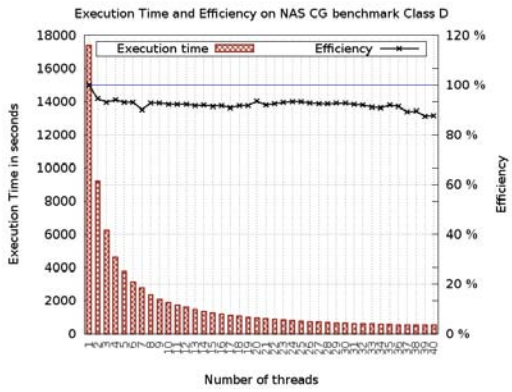
  Finally, we conclude that the scalability of the application is good, but class B presents a limitation of parallelism.



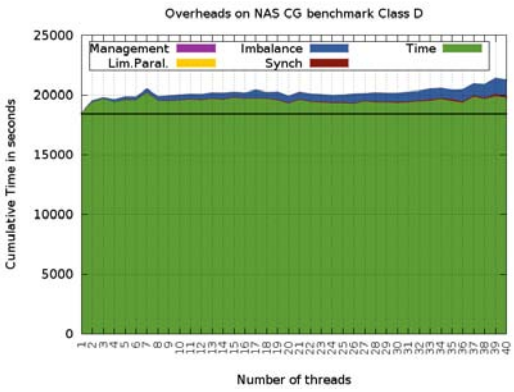
(a) Exection time an efficiency on CG.B



(b) Overhead analysis on CG.B



(c) Exection time an efficiency on CG.D



(d) Overhead analysis on CG.D

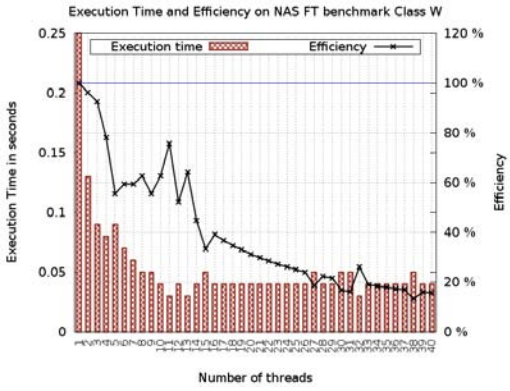
Figure 4.6: Scalability analysis on CG benchmark

- **FT (Fourier Transform)**; This benchmark contains the computational kernel of a 3-D fast Fourier Transform (FFT)-based spectral method. FT performs three one-dimensional (1-D) FFT's, one for each dimension.

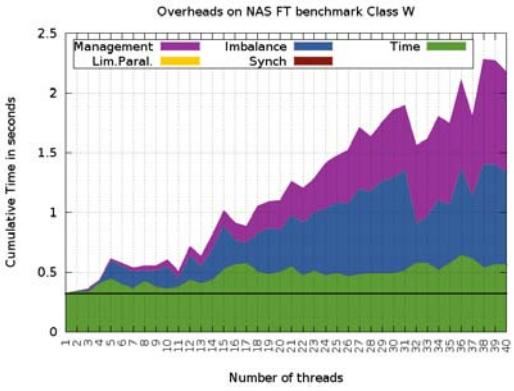
  Figure 4.7(a) presents the execution for the smallest workload with a limited scalability beyond 1 processor. That is mainly because of the NUMA effect. Figure 4.7(b) presents the significant impact of the management library for small execution times.

  The execution of class B, Figure 4.7(c) presents a more realistic workload configuration. However, efficiency starts decreasing from 80% beyond 1 socket execution up to 40% in a 40 threads execution.

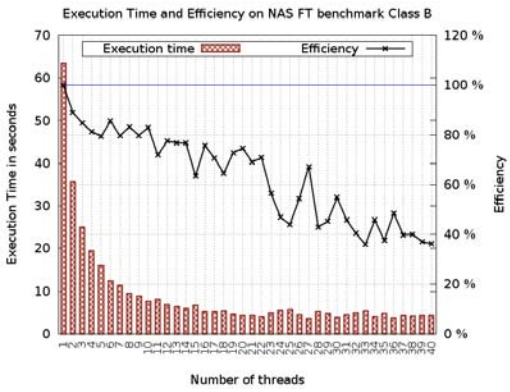
  The increase in per thread time shown in Figure 4.7(d) could be due to an imbalance on a multisocket execution or a limited parallelism for this system.



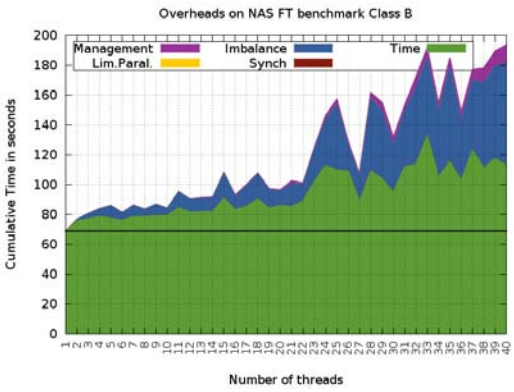
(a) Exection time an efficiency on FT.W



(b) Overhead analysis on FT.W



(c) Exection time an efficiency on FT.B



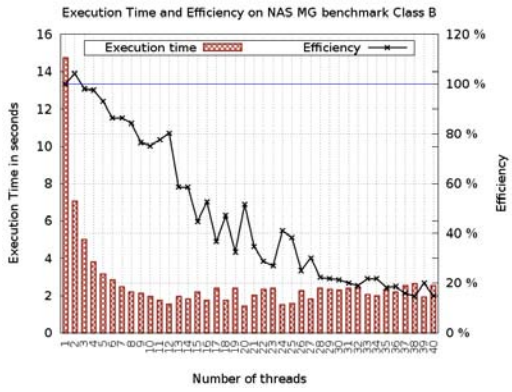
(d) Overhead analysis on FT.B

Figure 4.7: Scalability analysis on FT benchmark

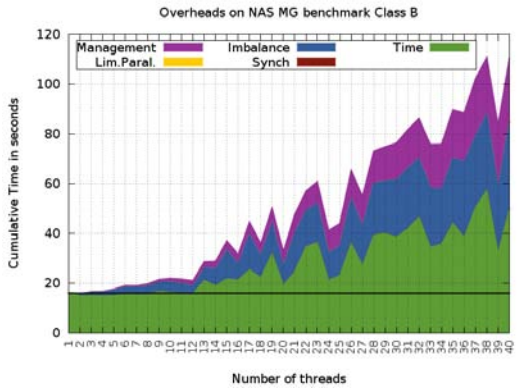
- **MG (Multi-Grid)**; The MG benchmark uses a V-cycle MultiGrid method to compute the solution of the 3-D scalar Poisson equation. The algorithm works continuously on a set of grids that are varying between coarse and fine. It tests both short and long distance data movement, and it is a memory intensive application.

  Figure 4.8(a) shows a limited parallelism for class B up to 1 processor as can be deduced by the impact on management overhead in Figure 4.8(b).

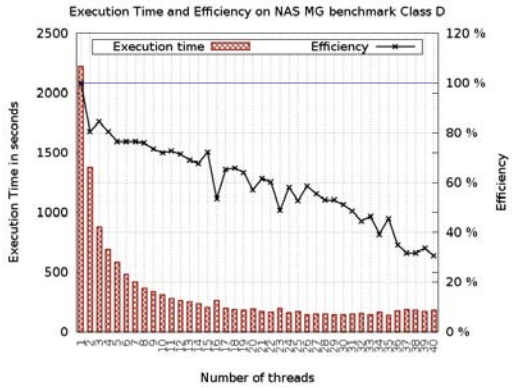
  The increased class D shows a better scalability and the efficiency drops more progressively. Figure 4.8(d) shows an increasing on execution time and imbalance, probably generated by the NUMA effect because it is defined as a memory intensive application.



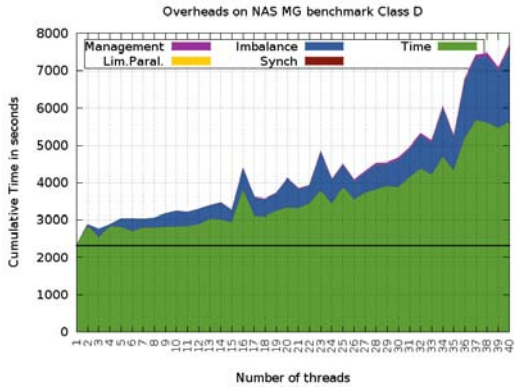
(a) Exection time an efficiency on MG.B



(b) Overhead analysis on MG.B



(c) Exection time an efficiency on MG.D



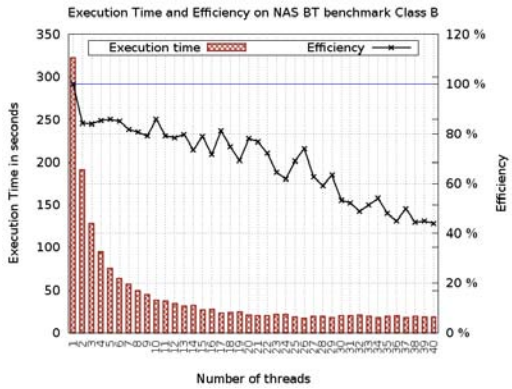
(d) Overhead analysis on MG.D

Figure 4.8: Scalability analysis on MG benchmark

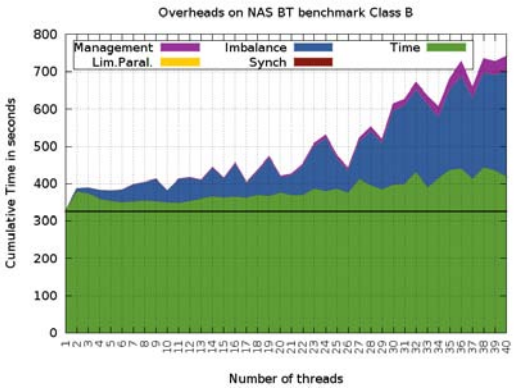
- **BT (Block Tri-diagonal)**; BT benchmark is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations.

  Performance for class B in Figure 4.9(a) shows a good scalability up to 20 threads, and a significant overhead and imbalance in Figure 4.9(b).

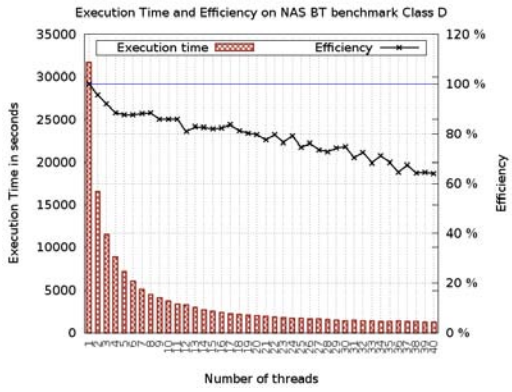
  Class D (Figure 4.9(c)) shows a better efficiency and scalability. The overheads in Figure 4.9(d) point to an increasing in execution time and imbalance which can be generated because of the NUMA effect.



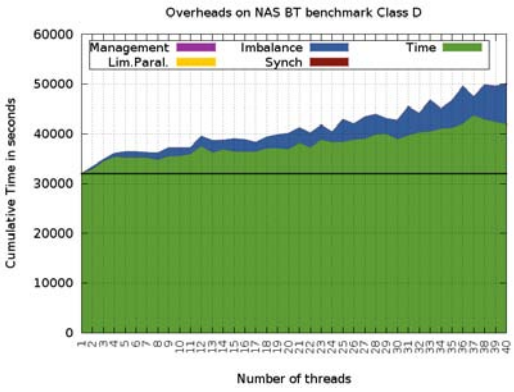
(a) Exection time an efficiency on BT.B



(b) Overhead analysis on BT.B



(c) Exection time an efficiency on BT.D



(d) Overhead analysis on BT.D

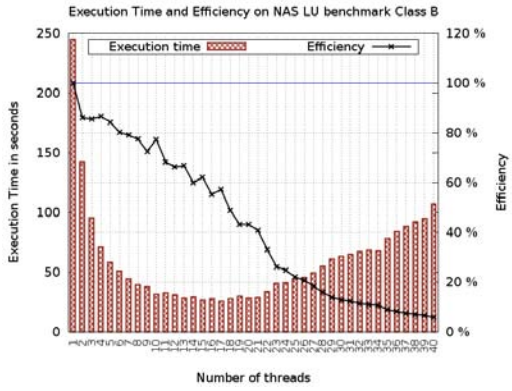
Figure 4.9: Scalability analysis on BT benchmark

- **LU (Lower-Upper Gauss-Seidel solver)**;

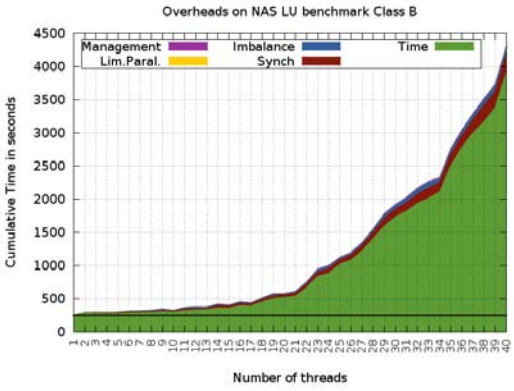
  LU is a simulated CFD application that uses a symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block Lower and Upper triangular systems.

  Figure 4.10(a) shows a significant performance degradation on class B with a sharp deterioration for more than 2 sockets. The cause of this degradation can be the limited parallelism and the syncronization elements as shown in Figure 4.10(b).

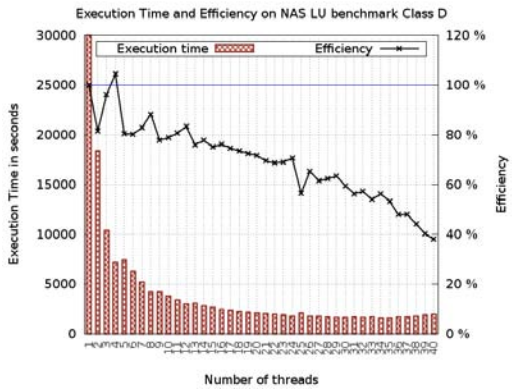
  For class D execution, the application shows a better scaling, Figure 4.10(c), but a drop on efficiency when using more than 2 sockets. Figure 4.10(d) shows that the impact of this contention can be generated by the utilization of synchronization elements.



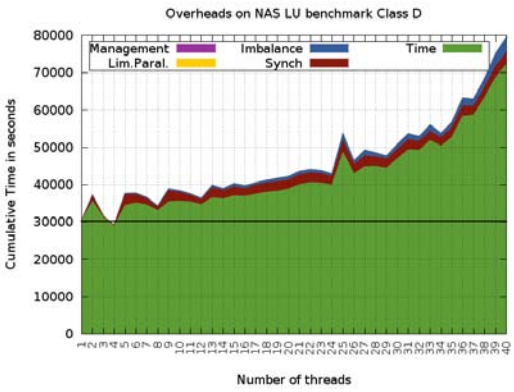
(a) Exection time an efficiency on LU.B



(b) Overhead analysis on LU.B



(c) Exection time an efficiency on LU.D



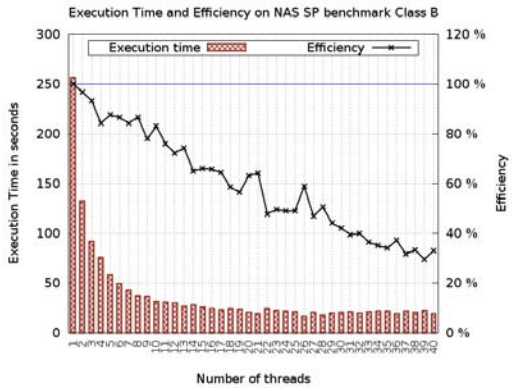
(d) Overhead analysis on LU.D

Figure 4.10: Scalability analysis on LU benchmark

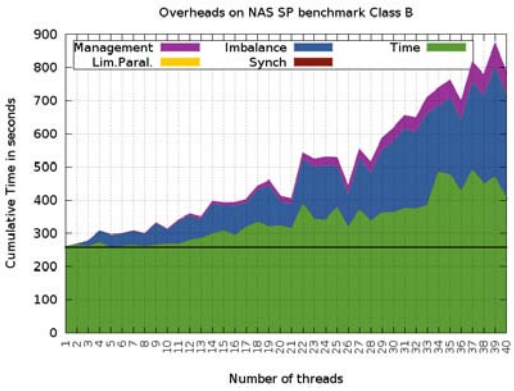
- **SP (Scalar Penta-diagonal)**; SP is a simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has Scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension.

  Figure 4.11(a) shows performance for class B, with a limited scalability for more than 1 socket, were efficiency goes from 80% to 20%. Figure 4.11(b) shows an increasing overhead in management and imbalance, and also execution time per thread.

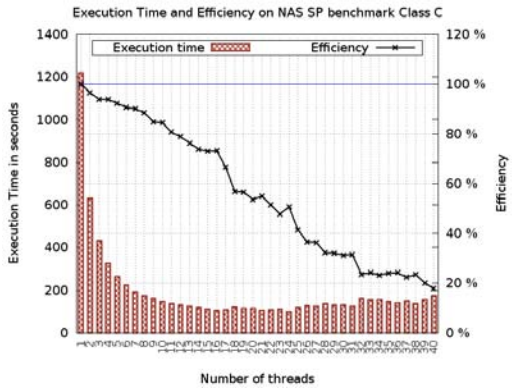
  The class C execution in Figure 4.11(c) shows a performance contention around 16 threads, and performances degrades in the full thread execution. Figure 4.11(d) shows a pattern of increased overhead around the 20 threads execution.



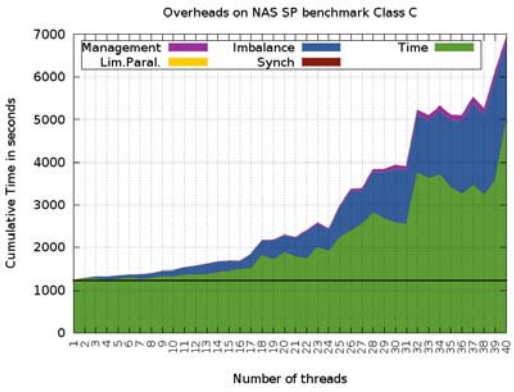
(a) Exection time an efficiency on SP.B



(b) Overhead analysis on SP.B



(c) Exection time an efficiency on SP.C



(d) Overhead analysis on SP.C

Figure 4.11: Scalability analysis on SP benchmark

Discussion

Performance and scalability have been characterized on the NAS parallel benchmarks. It can be observed that for large workloads performance and scalability in EP, CG and BT is good. In this environment, the evaluation of efficiency for large workloads in FT and LU is 40%, MG is 30% and finally on SP 20%.

From this results, we conclude that it is necessary to evaluate the benchmark with higher performance degradation (SP), by replicating the performance problem on a different system architecture, in order to justify the necessity of identifying and model the associated performance factor.

4.2.2 Analysis of performance factors

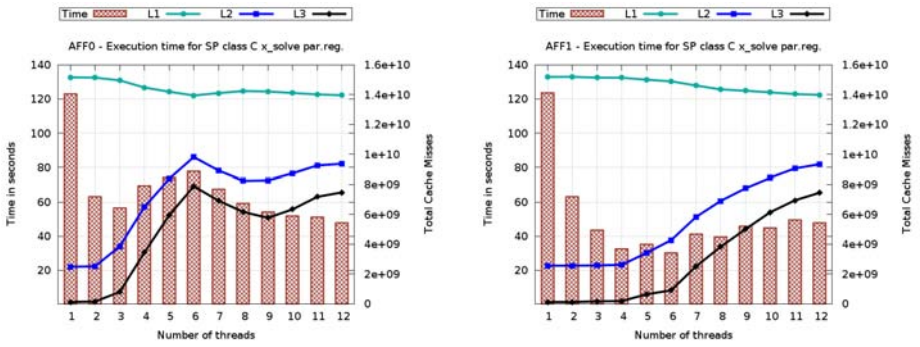
A deeper analysis is done for the SP application using a different system architecture and different workloads. The experimental environment is the T7500 system (DELL workstation). This shared memory system is a dual processor system with 6 cores per processor, providing up to 12 threads per node (no Hyperthreading). The execution has been profiled with ompP, to obtain a fine grained detailed report for parallel regions within the application, and metrics from different hardware counters have been considered.

In the following experiments, threads are binded to cores using the likwid-pin tool. There are two different thread scheduling policies, the compact distribution policy (#AFF0) schedules threads using the minimum number of different processors, and a scattered distribution policy (#AFF1) which distributes threads along the processors.

The performance analysis is focused on the x_solve parallel regions, which is the parallel regions with the higher degree of performance degradation. The execution time for the x_solve parallel region is shown in Figure 4.12(a). A performance degradation can be seen on the parallel execution on 1 processor (from 1 to 6 threads). However, the full thread execution obtains the best performance in this case.

On the other hand, the execution of a scattered distribution policy (#AFF1) shows a completely different performance. In this case, the best performance is obtained with half of the threads available (6 threads), which in this policy represents 3 threads per processor. A performance degradation appears when using more than 6 threads.

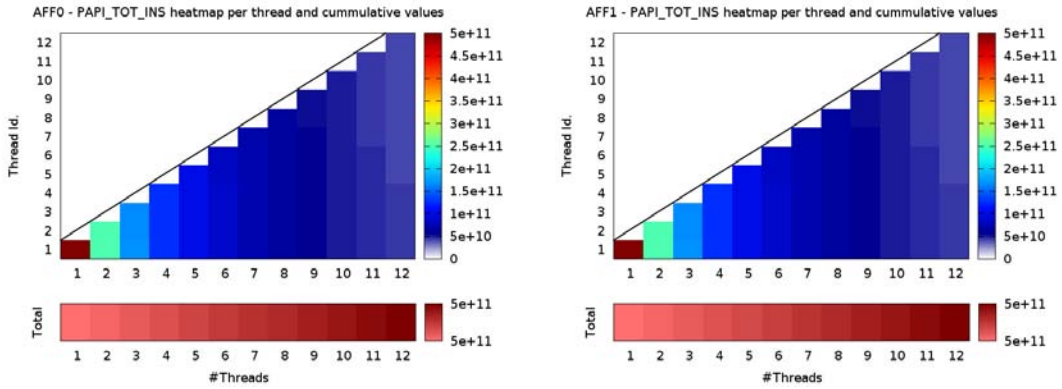
The hardware subsystem is evaluated by monitoring the total cache misses for the three levels of cache, and results presented as the overlapped lines in Figures 4.12(a) and 4.12(b). At this point, we consider the hypothesis that the behaviour of total cache misses seems to be correlated with the execution time, but a further analysis needs to be done.



(a) Compact distribution (AFF0) (b) Scattered distribution (AFF1)

Figure 4.12: Total cache misses and execution time on parallel region x_solve for SP.C

To analyze the hardware implications on the performance degradation of this application we start analyzing the number of total instructions shown in Figures 4.13(a) and 4.13. Each figure contains two relevant plots. The diagonal matrix is a heatmap representation of total instruction per thread. For example, in a three thread execution (x-axis), we have a hardware counters value for each thread (y-axis). The second plot, at the bottom of the figure, represents a heatmap of cumulative values for the counters, in this case, total number of instructions.

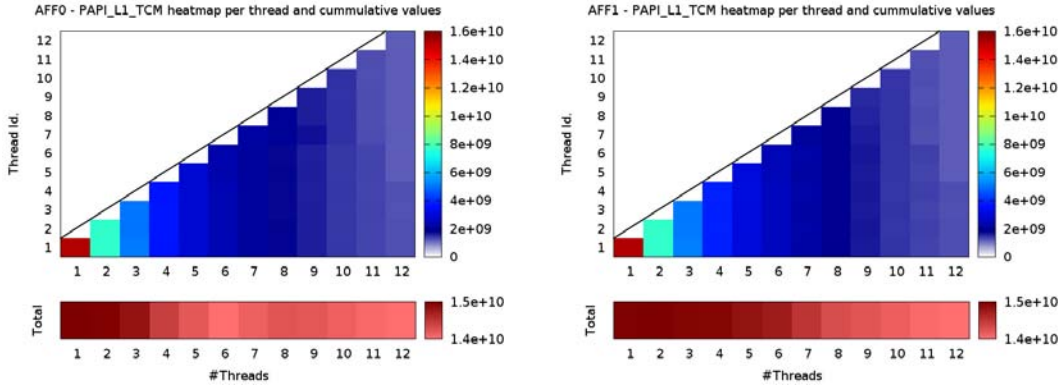


(a) AFF0 – Total Instructions (b) AFF1 – Total Instructions

Figure 4.13: SP.C x_solve – Heatmap of hardware counter Total Instructions

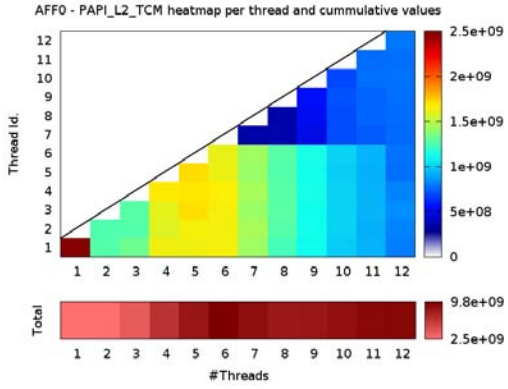
In both figures, the diagonal matrix presents an almost regular gradient, meaning that the total instructions are evenly distributed for all the thread configurations. The scale of the heatmap in the second plot presents a gradient between the minimum and maximum values, which are same scale values. Therefore, we can assume that the work is evenly distributed among threads, and this is not a performance factor.

Following this, the same representation is used in Figures 4.14(a) to 4.14(f) to evaluate cache misses for the different levels of the cache hierarchy.

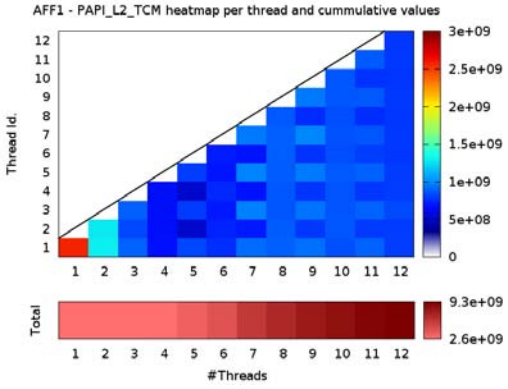


(a) AFF0 – L1 TCM

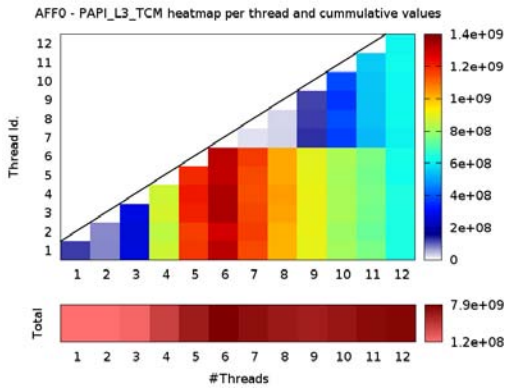
(b) AFF1 – L1 TCM



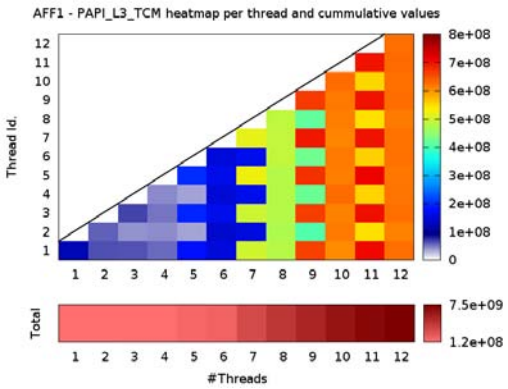
(c) AFF0 – L2 TCM



(d) AFF1 – L2 TCM



(e) AFF0 – L3 TCM



(f) AFF1 – L3 TCM

Figure 4.14: SP.C x_solve – Heatmap of hardware counter Total Cache Misses

Figures 4.14(a) and 4.14(b) do not present a significant differentiation. Total cache misses in L1 remains almost constant, as can be seen in previous figures 4.12(a) and 4.12(b).

However, Figures 4.14(c), 4.14(d), 4.14(e) and 4.14(f) present a differentiate pattern in thread behaviour. The compact (AFF0) figures, present a hotspot in the six thread execution which is equivalent to the 1 processor execution, and this hotspot slightly tends to relax in a gradient until the cache misses is balanced between the two processors. There is a visual differentiation between the workloads of the two processors.

Furthermore, the scattered distribution can also be visualized on the right size hotspot figures. The interleaving on number of threads in the odds configurations indicates its natural imbalance. On the contrary, even number of threads distributions remain well balanced.

The scattered distribution, on the right side of the figures, is displacing the hotspot to the most concurrent configurations.

Discussion

We conclude from this analysis that a significant performance problem exists in SP benchmark, and performance degradation can be avoided. To do that, it is possible to define an appropriated thread scheduling policy and number of threads. The associated performance factor is based on **performance degradation** due to **memory contention**, which is caused by the over utilization of the last levels of cache in a single processor execution at certain degree of concurrency.

From this, we propose to model the associated performance factor and evaluate a strategy to dynamically characterize at runtime the different configurations of number of threads to identify the best configuration of number of threads for a parallel region. By doing this, for all the parallel regions, the overall performance can be improved.

4.3 Evaluating a strategy for tuning the number of threads

By considering the previously described performance factor candidate on the Scalar Pentadiagonal solver (SP) application of the NAS Parallel Benchmark suite, we proceed to apply the second part of the methodology to model the performance factor and define a strategy for dynamically tune the number of threads.

To do that, we evaluate the context to expose the performance problem on two new different architectures with the aim of validating the persistence of the performance factor for different systems.

In this section we are going to define the new context, propose an initial approach for performance modeling, evaluate the impact of performing a static strategy and finally, to evaluate a dynamic tuning strategy.

4.3.1 Context

The experimental environment has been provided by the Universitat Oberta de Catalunya (UOC) and the CAOS department at the UAB, and it is composed of two different cluster nodes environment.

System characterization

Taking into consideration that the candidate performance factor is memory contention, especial emphasis is made on the description of the hardware memory subsystem and the application memory related parameters.

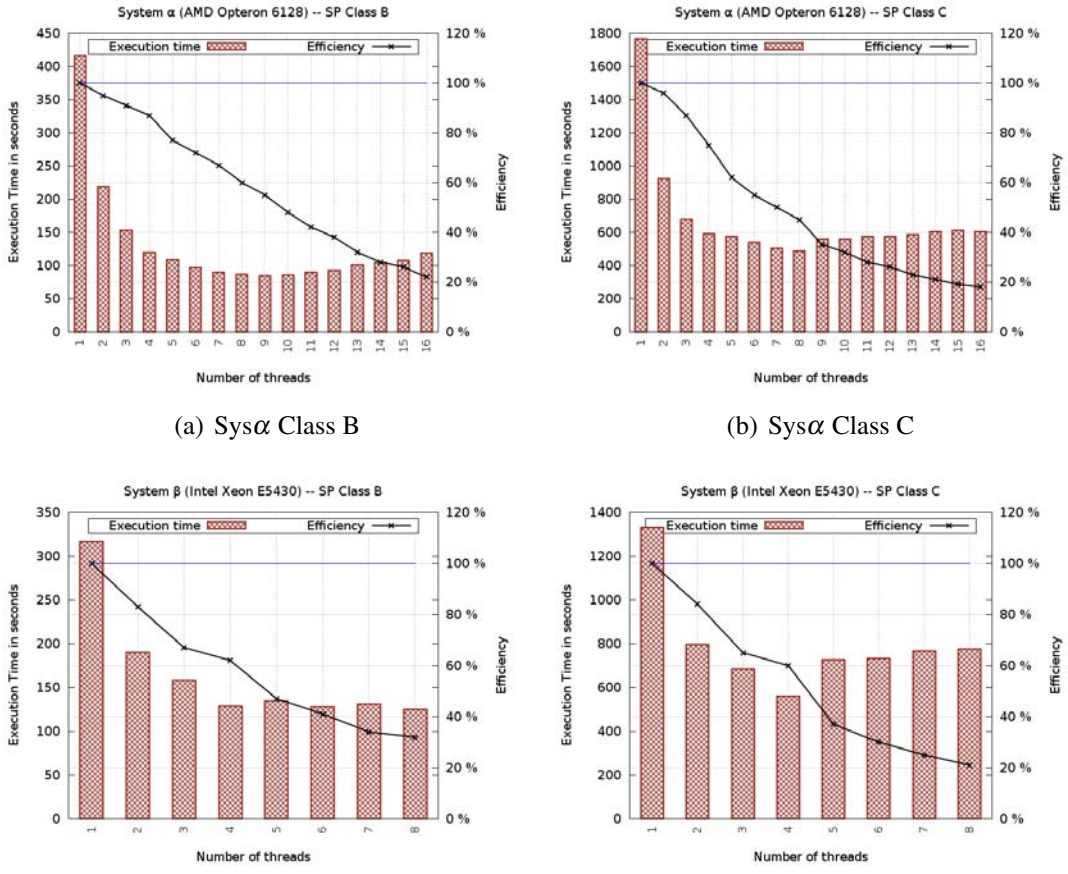
- **Application**: Experimental evaluation of SP benchmark of NPB-3.3.1 suite (OpenMP version), for classes B and C. Problem workload sizes are based on cubic structures of 102 dimension elements on class B and 162 dimension elements on Class C. The algorithm performs an iterative factorization process, and convergence is evaluated after 400 iterations, and compared with a precalculated solution. This implementation decouples the x, y and z dimensions and provides a solver function for each dimension.
- **Manager library**: Both systems uses the GCC compiler 4.5.2, which provides the libgomp OpenMP runtime library implementing OpenMP 3.1 specification.
- **Hardware**:
  - Sysα; NUMA system with 2x AMD Opteron 6128 @ 2GHz processors with 8 cores; total amount of 16 cores and 32GB of main memory. Shared cache L1 of 64KB per core pair, Cache L2 of 512KB shared by core pairs and shared 5MB of Cache L3 by groups of 4 cores.
  - Sysβ; NUMA system with 2x Intel Xeon E5430 @ 2,6GHz processors with 4 cores; total amount of 8 cores and 16GB of main memory. Dedicated Cache L1 of 32KB and 6MB Cache L2 shared by core pairs.

Analysis of performance factors

For the analysis of the performance factor, we proceed to configure the environment for the classes B and C and perform a strong scalability analysis to identify the possible performance degradation in systems Sysα and Sysβ.

Firstly, we have to notice that cache sizes on these systems are noticeable different to the previous ones. On the one hand, Sysα has 5MB of L3 cache shared by groups of four cores, and Sysβ has 6MB of L2 cache shared by groups of two cores.

In the previous evaluation, Figure 4.11(a), the scalability analysis for class B reported a performance contention and the analysis of overheads in Figure 4.11(b) showed an increasing imbalance and management overhead.



(a) Sysα Class B

(b) Sysα Class C

(c) Sysβ Class B

(d) Sysβ Class C

Figure 4.15: Scalability analysis for SP Classes B and C for systems Sysα and Sysβ

Figure 4.15(a) for Sysα shows the scalability analysis for class B and Figure 4.15(b) for

class C. A performance degradation can be observed. By using half of the available threads the application achieves its best performance.

For Sysβ, Figure 4.15(d) also expresses the performance degradation but in Figure 4.15(c) the best performance is obtained with the maximum threads configuration. However, the difference between the 4 thread and 8 thread executions is less than 4 seconds, 128.34 vs. 124.86 respectively.

We have traced 10 applications iterations for Sysβ using a configuration of 4 threads(559 seconds) and 5 threads (725 seconds). Figure 4.16 and Figure 4.17 show the execution timeline for one iteration. The information provided by Paraver consists on execution time represented by dark blocks and the light blocks represent overheads due to synchronization, scheduling, and idle states.

Using the tracing analysis, the execution imbalance for the 5 thread configuration can be observed. The previous evaluation using the ompP profiler indicated an increasing imbalance and management overhead on the multiple socket execution, and Paraver also reports the same information.

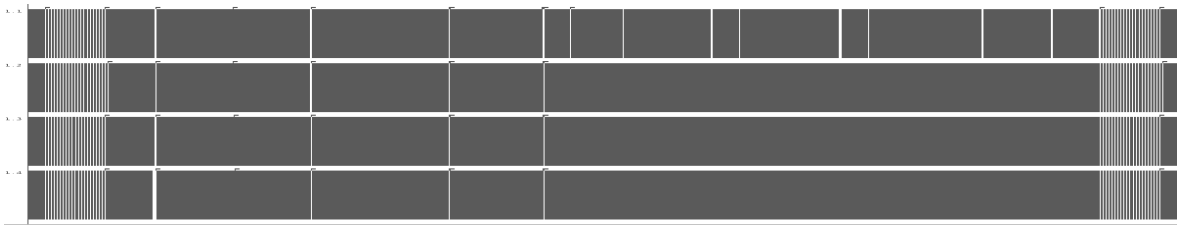


Figure 4.16: System Sysβ (4 threads) - 1 SP.C iteration

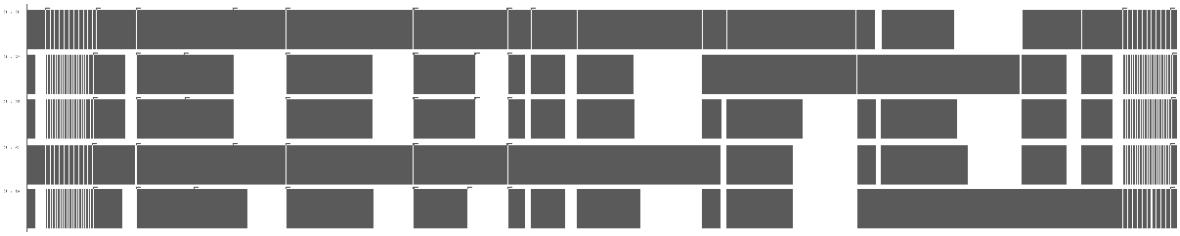


Figure 4.17: System Sysβ (5 threads) - 1 SP.C iteration

The profiling based on trace analysis shows that the summarized imbalance for 4 threads configuration is less than 1%, but for the 5 threads execution it raises to 14%.

The application is analyzed at parallel region level for the 4 most representative functions in SP. The percentage of the execution time expended in each parallel region is presented in Table. 4.1. This table shows that the four main parallel regions represent about the 80% of the execution time of the application.

Table 4.1: Sysβ Class C execution time (sec.) and cumulative percentage (relative to total time T_{ref}) of use for the weightiest parallel regions (x,y and z_solve, and rhs).

	Number of threads								
Parallel Region	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	%degradation
x_solve	175	87	60	**46**	85	88	103	**120**	160%
y_solve	199	100	70	**53**	87	90	99	**106**	100%
z_solve	224	112	79	**60**	94	96	105	**113**	88%
rhs	537	327	331	**277**	328	342	330	**315**	13%
T_{ref}	1,331	797	683	559	725	732	765	777	38%
%	85.2	82.8	79.3	78.3	81.8	81.9	83.2	84.4	

The performance degradation between the 4 thread execution in comparison with the 8 thread execution, of x_solve(160%), y_solve(100%), z_solve(88%) and the rhs(13%), is quite significant.

Next, we proceed to analyze hardware performance implications for the most degraded parallel region (x_solve) using PAPI hardware counters for the L2 cache misses percentage $\%C_{L2}$ in a single iteration in Sysβ for the class C.

Table 4.2 shows that the L2 cache misses ratio for 8 threads is 16 times higher than for 4 threads.

Table 4.2: x_solve parallel region on Sysβ Class C for one iteration execution. Where T_{it_n} is the time for n-iteration and T_{ref} is the measured time for 400 iterations.

Parallel Region	**Number of threads**							
x_solve	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
ratio of L2 cache misses	1	1	1	1	**7**	**11**	**14**	**16**
First Iteration execution time	0.45	0.25	0.16	0.13	0.22	0.22	0.27	0.3
Single sample Estimation(x400)	180	100	64	52	88	88	108	120
Total execution time (400 *iters.*)	175	87	60	46	85	88	103	120
Relative error	2.8%	14.9%	6.6%	13%	3.52%	0%	4.8%	0%

In this case, the concurrency within the same cache level is not generating a performance degradation, however, the performance is degrade when using different sockets.

We conclude, that in some cases the performance degradation is generated due to cache

memory over utilization (such as FatNode system in Section 4.2.2), and also can be expressed due to the NUMA effect, such as in Sysβ.

4.3.2 Modeling performance and defining tuning strategies

We have observed that a performance factor based on performance degradation can be expressed in a different way depending on the system architecture. On the one hand, a performance degradation is expressed at certain degree of concurrency at processor level when sharing the same LLC (e.g. FatNode System) and, on the other hand, some architectures express performance degradation when using more than 1 processors in NUMA systems (e.g. Sysβ System).

We have shown in Table 4.1 that in the SP application the most significant performance degradation (%degradation) occurs in the x_solve parallel region, and Table 4.2 shows that the estimation based on single application iteration achieves an acceptable degree of accuracy compared with the full execution time, with a maximum relative error of 15%.

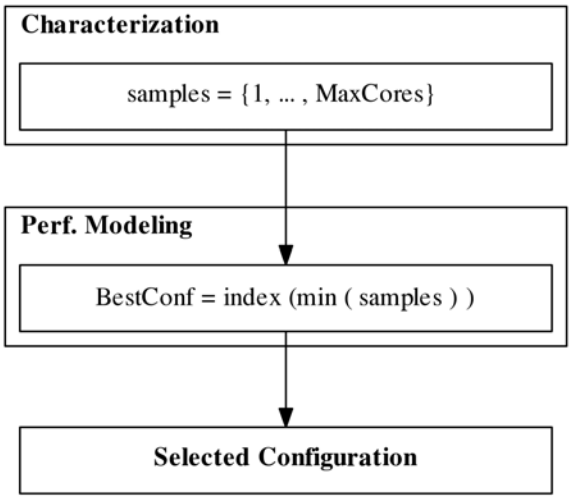


Figure 4.18: Methodology to select the configuration providing the minimum execution time based on an exhaustive runtime characterization on the first iterations of the application.

Figure 4.18 presents an initial performance model based on sampling the first iterations of the application using different configurations of number of threads in order to determine the overall performance (expression 4.1). By doing this, it is possible to obtain a configuration that minimizes performance degradation (equation 4.2).

$$samples = \{time_i, time_{i+1}, .., time\_MaxCores\} \quad , where \quad i \in 1..MaxCores \tag{4.1}$$

$$BestConf = i \quad , where \quad samples_i = MIN(samples) \tag{4.2}$$

The number of threads can be dynamically tuned to the $BestConf$ value by instrumenting the application and inserting previously to the definition of the OpenMP parallel region a `omp_set_num_threads(VALUE)` library function call, which explicitly defines the number of threads of the work-sharing construct for the subsequent parallel region. Taking advantage of this previous knowledge, an initial coarse grained strategy is presented.

The SP application is iterative, therefore, it is possible to invest some iterations to acquire a prior knowledge of the performance. A fraction of the 400 iterations required to compute the solution in classes B and C can be used to characterize its performance. Following this, the best configuration can be selected to tune the application for the remaining iterations. In order to reduce the overheads, the number of iterations invested in characterizing the performance must be as small as possible.

4.3.3 Evaluating the impact

To evaluate the impact, the strategy has been implemented in the application source code. The class C on Sysβ reports an execution time of 562 seconds, which represents only an overhead of 0.5% compared to the 559 seconds of the best configuration execution using 4 threads.

This strategy requires a characterization based on an exhaustive exploration of all the possible configurations of number of threads. Furthermore, to obtain a more precise accuracy every sample can be averaged from different iterations of the application for the same configuration. Therefore, the overhead of this strategy is going to depend mainly on the characterization phase. Furthermore, it could happen that the overhead generated on the characterization phase would exceed the improvement in performance.

The analysis of performance factors have shown that the cache behaviour due to memory occupancy at cache level, and the thread distribution within the NUMA environment for some configurations of number of threads can generate performance contention and performance degradation. Therefore, by defining a performance model identifying these critical configuration would reduce the current overheads generated on the exhaustive characterization, and provide a more scalable strategy.

Algorithm 1 Pseudocode of the tuning strategy applied by using the linker preload mechanism for the interposition of the OpenMP dynamic library. The strategy performs time characterization on the firsts iterations. The retuning value of function *characterization_Phase* is based on a user defined threshold determining the number of iterations used for characterizing the application (greater than NC).

```
Function GOMP_parallel_start(params)
1   if characterization_Phase(appIter) then
2       config_Params = generate_Config(appIter)
3       Start_Timer()
4       real_GOMP_parallel_start (config_Params)
    else
5       real_GOMP_parallel_start (tuning_Params)
    end
End Function
Function GOMP_parallel_end()
1   real_GOMP_parallel_end()
2   if characterization_Phase(appIter) then
3       Stop_Timer()
    else
4       if last_charact_phase (appIter) then
5           tuning_Params = time_Analysis()
    end
6   appIter++
End Function
```

4.3.4 Applying the dynamic tuning strategy

With the aim of demonstrating the benefit and feasibility of dynamically tuning the OpenMP scheduling library, and to analyze the tuning overheads, we have implemented the rough strategy described in the previous section by using a dynamic tuning strategy.

To do that, we have selected a technique based on a library interposition of the OpenMP runtime library at linking time.

The most relevant parallel functions within the SP application use parallel regions and parallel loops with static scheduling. The low level templated transformations performed at compilation time defines the entry and exit points as function calls to the runtime library. The interception of these function calls is used to embed the monitoring, the analysis and the tuning required for the strategy. The algorithm for the dynamic tuning strategy is defined in Algorithm 1.

The dynamic tuning strategy has been evaluated in the SP benchmark for Sysα and Sysβ.

During the characterization phase 5% of the total number of iterations is used to test the candidate thread configurations. The speedup obtained with the tuning strategy is shown in table 4.3.

Table 4.3: Execution time (sec.) for the dynamic tuning strategy and execution without tuning for classes B and C. The tuning strategy uses 5% of total iterations for the characterization stage.

Class	**Tuning**	**Sysα.**	**Sysβ.**	**Class**	**Tuning**	**Sysα.**	**Sysβ.**
B	max. threads	118	124	C	max. threads	604	777
B	Dynamic	128	86	C	Dynamic	485	586
Speedup		0.92	1.44	*Speedup*		1.24	1.32

The results on Table 4.3 shows a slowdown of 0.92% for class B on Sysα. This is because the best configuration for class B uses the maximum number of threads, and the performance slowdown is generated on the characterization an instrumentation phases. However, on the rest of the experiments the best configuration uses half of the threads and the speedup is improved by the tuning strategy. Sysβ for class B achieves the best speedup of 1.44x.

4.4 Summary

In this chapter we have evaluated the OpenMP version of the NAS parallel bencharmk suite, a representative set of kernels and pseudo application mimicking the behaviour or real HPC applications. The performance evaluation shown in the methodology presents strong scalability analysis on different system architectures, from systems with a small scalability to highly scalable environments. The performance evaluation on such systems has shown a good performance in all context for EP, BT, and performance contention or degradation on some scenarios for CG, FT, MG, LU, SP in some cases due to limited parallelism for the context workload, and in other cases due to hardware architecture characteristics.

Through the analysis of SP benchmark by using performance counters we have deeply analyzed a memory contention problem on a single socket execution, and the alleviation of the memory contention by spreading the execution units along a multisocket system obtaining a better speedup compared with the execution using all the execution units.

Following this, we have tried to extrapolate this problem from the context of the evaluated system architectures, and we have analyzed two different systems (Sysα and Sysβ) with the

aim of replicate the same performance problem on a different context. These systems have also express the performance problem, also for class B in Sysβ.

In order to prevent the performance degradation, we have proposed and evaluated a dynamic tuning strategy based on characterizing the first iterations of the application by using different configuration of number of threads and selecting the best configurations by measuring the execution time for each parallel region.

The tuning strategy evaluated has provided the best configuration in all cases, with a negligible overhead. However, we have pointed that the initial strategy requires an exhaustive characterization, and we expect that the characterization overheads will increase on more scalable systems architectures.

In order to provide a better detection of the performance problems, we suggest a new iteration on the methodology in order to provide a performance model with smaller overheads.

To do that, in the following chapter we define a performance model based on the estimation of the execution time for a reduced number of configurations. This model is focused on the analysis of hardware counters, because the hotspot analysis of hardware counters has shown a relation between performance contention at last levels of cache and performance degradation.

5

Performance model based on runtime characterization

"Ne marche pas devant moi, je ne suivrai peut-être pas. Ne marche pas derrière moi, je ne te guiderai peut-être pas. Marche juste à côté de moi et sois mon ami."

Les justes - **Alber Camus**

This chapter introduces a performance model based on runtime characterization on a single socket which allows to estimate a configuration of number of threads and thread distribution to improve performance on a multicore multisocket system.

5.1 Introduction

Following the previous chapter, we have defined a tuning strategy based on an exhaustive characterization at runtime, however, the overhead generated on the characterization phase can significantly impact performance. To reduce the characterization phase, we propose a characterization of the performance of parallel regions to estimate cache misses and execution time by limiting the characterization phase to an exhaustive execution on a single socket.

Furthermore, this model is used to extrapolate performance for different number of threads and different affinity distribution for each parallel region in a multisocket system. The model is applied for SP and MG benchmarks from the NAS Parallel Benchmark Suite using different workloads on two different multicore, multisocket systems. These benchmarks have been selected because they are memory intensive applications, and the previous evaluation of NAS benchmark in Section 4.2.2, they have shown the smallest degree of efficiency, being 20% for SP, and 30% for MG.

As we will see in Section 5.5.1, the estimation preserves the behavior shown in measured executions for the affinity configurations evaluated. Section 5.5.2, shows how the estimated execution time is used to select a set of configurations in order to minimize the impact of memory contention, achieving significant improvements compared with a default configuration using all threads.

This chapter is structured as follows. Section 5.2 introduces the objective and scope of this model. Section 5.3 introduces related work about analytical performance modeling. Section 5.4 introduces our performance model for estimating total cache misses (TCM) at the last level cache (LLC) and estimated execution time. The model is validated in Section 5.5, where it is evaluated using the SP and MG benchmarks for two different architectures. Finally, Section 5.6 summarizes our conclusions.

5.2 Objective

Performance on shared memory systems must consider multicore multisocket environments, with different sharing levels of resources in the memory hierarchy. To take advantage of shared memory systems, the high performance computing community has developed OpenMP Application Program Interface (OpenMP) defining a portable model for shared-memory parallel programming. However, depending on the memory utilization, the memory interface can become a bottleneck. It is possible to group threads to take advantage of sharing memory

or, on the other hand, distribute them in the memory hierarchy or restrict their number to avoid degradation due to memory contention.

To this aim, we propose a performance model based on characteristics of the multicore multisocket architectures and the application memory pattern. The model estimates the runtime of an application for a full set of different configurations in a system regarding the thread distribution among cores (affinity) and number of threads. The model is evaluated using runtime measurements on a partial execution of the application in order to extract the application characteristics.

To develop our approach we have made the following assumptions:

1. The application is iterative and all iterations have uniform workload;
2. Workload is evenly distributed among threads;
3. Performance degradation is mainly generated by memory contention at LLC
4. All processors in the socket are homogeneous. Our input parameters for the model are based in the measurement in a single socket execution.

Taking into account these assumptions, our contributions are the following:

- A performance model to estimate the LLC misses for different affinities at the level of individual parallel regions.
- A performance model to estimate the execution time for a parallel region, considering an empirical value to adjust the parallelism degree at the memory interface level and data access pattern.

5.3 Related work

There are several approaches to estimate shared memory systems performance. Tudor [75] presents a performance analysis for shared memory systems, and a performance model. It is validated with NAS parallel benchmarks. This model considers idleness of threads, which is present in context switching, specially when more than one thread per core is executed. We consider our model to focus on the cache behavior because memory contention is the main cause for performance degradation in memory bound HPC applications.

We use the idea of performance degradation in the context of parallel executions. Following this, [76] presents the impact of cache sharing. The analysis is based on the characterization of applications on isolated threads and, Zhuravlev in [77] presents two scheduling algorithms to distribute threads based on miss rate characterization. Dwyer et al. [78] present a practical method for estimating performance degradation on multicore processors. Their analysis is based on machine learning algorithms and data mining for attribute selection of native processor events. We also obtain information from performance hardware counters but without using database knowledge obtained on a postprocessing analysis, that information is obtained by using empirical data from a reduced sample of data that could be achieved at runtime.

Regarding the hardware, the Roofline model [69] is a visual computational model to help identifying applications characteristics such as memory bound limitations. This model shows how operational intensity can provide an insight of architecture and application behavior, and provides an insight of the architecture, however this model is oriented to help development and provide suggestions to make code optimizations on the source code. In our case, we present a model in order to select an affinity configuration with the aim of being used at runtime in an automatic tuning tool.

5.4 Performance Model proposal

Performance degradation in memory bound applications considered in this work can be produced depending on application data access pattern and its concurrency at cache level. Therefore, characteristics such as workload and data partitioning, the degree of data reutilization of the data access pattern based on temporal and spatial locality, data sharing between threads, and data locality on the memory hierarchy must be considered. Consequently, a deep knowledge of the application behavior and system architecture to improve performance is required.

Iterative applications can provide similar performance among iterations as we have shown in section 4.3 from previous chapter. For this case, it is possible to apply a strategy (Figure 5.1) to evaluate the behavior of the application for a reduced set of iterations with different configurations regarding the degree of parallelism and thread pinning configurations. Our model considers these measurements to estimate the execution time for the total set of configurations in the system.

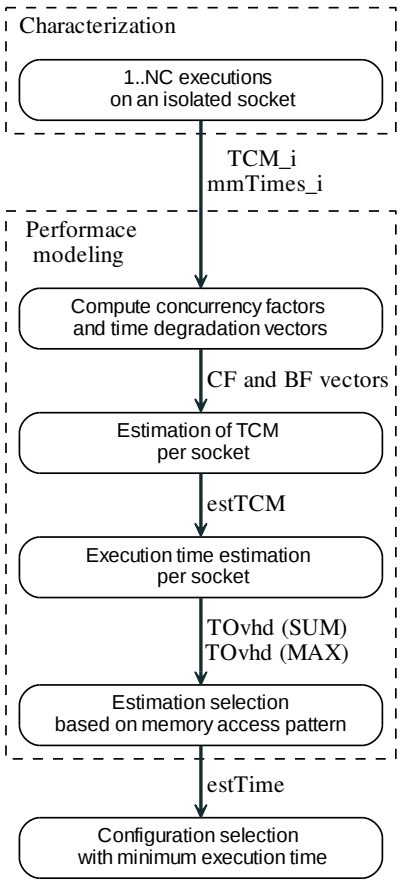


Figure 5.1: Methodology for the selection of the number of threads and its affinity distribution.

5.4.1 Defining the performance model

In order to apply the proposed model, *NC* executions with parallel region profiling are required, *NC* being the number of cores in a single socket, the i-th execution runs on threads 0 to $i-1$. This allows us to obtain the model's input parameters for time and hardware counters (LLC_MISSES). We consider that ideal run time is mainly altered by memory contention at shared cache level, and this contention is measured by the *LLC_MISSES* hardware counter, which provides the number of inclusive miss events at LLC for the system architecture, meaning that the data is not present on the socket and must be acquired on memory. We collect the total cache misses (TCM) generated at last level cache for each parallel region in order to analyze the concurrency overhead.

The parameters involved in our model are described on Table 5.1.

Table 5.1: Table of parameters used to estimate the execution time of N threads for a given configuration.

Parameter	Description
NC	the number of cores in a socket
NS	the number of sockets in the system.
cf_i	concurrency factor for i threads in a socket. It is expressed as TCM rate for the i execution over 1 thread execution. Being $i \geq 1$ and $i \leq NC$. these coefficients are measured at runtime for an isolated socket.
CF	NC size vector containing cf_i concurrency factor coefficients.
β_i	time degradation for i threads measured in a single socket. Being $i > 1$ and $i \leq NC$.
BF	NC size vector containing β_i factor coefficients.
aff_s	number of threads in the s-th socket for an AFF configuration.
AFF	affinity configuration, described as an NS size vector containing the specific number of threads per each socket for a given configuration.
$estTCM$	estimated TCM on the s-th socket for the AFF configuration.
$TOvhd$	estimated overhead time for the s-th socket on the execution of the AFF configuration
$mmime$	measured execution time.
$idealTime$	estimated ideal execution time.
$estTime$	estimated execution time.

Model input parameters

We propose to measure performance degradation on an isolated socket. Therefore, the model considers two known elements, the increase of TCM on a single socket due to concurrency, and its overhead time (taking into account the parallelism at memory level).

Concurrency behavior in a single socket at last level cache is represented by the vector (CF) of concurrency factors, defined in expression 5.1.

$$CF = \{cf_1, cf_2, ..., cf_i\}, \qquad \text{where } i \in 1..NC \tag{5.1}$$

Where each cf_i is the relation, defined in expression 5.2, between the measured TCM for a 1 thread execution and the measured cumulative TCM for an execution with i threads in the socket. This vector can be generated for each parallel region.

$$cf_i = \frac{TCM_i}{TCM_1}, \qquad \text{where } i \in 1..NC \tag{5.2}$$

On the other hand, to estimate the overhead time generated on memory accesses, we must consider that the memory interface is capable of achieving a degree of parallelism resolving the access requests. The full utilization of parallelism depends on the application data access pattern. Therefore, to express the relation between the achieved memory parallelism on a single socket and the application behavior, we define the vector (BF) of β factors in expression 5.3. These values are also obtained with the measured values in a single socket execution.

$$BF = \{\beta_1, \beta_2, ..., \beta_i\}, \qquad \text{where } i \in 1..NC \tag{5.3}$$

Each β_i factor (defined by 5.4) represents, for the i threads execution in a socket, the relation between the measured time ($mmTime$), and the overhead for the worst case scenario, providing a ratio of memory parallelism. The worst case is a serialized data miss access with no memory parallelism, implying a latency overhead per data miss. Also, we consider ideal time ($idealTime_i$) as $\frac{T_1}{NT_i}$, being T_1 execution time for 1 thread, and NT_i the number of thread for the i-th execution.

$$\beta_i = \frac{mmTime_i - idealTime_i}{TCM_i}, \qquad \text{where } i \in 1..NC \tag{5.4}$$

Following this, to represent the set of possible configurations in a system with NS sockets, the thread configuration is represented in expression 5.5 as the affinity vector AFF, where each component represents the number of threads in the $s-th$ socket.

$$AFF = \{aff_1, aff_2, ..., aff_s\}, \qquad \text{where } s \in 1..NS \tag{5.5}$$

The maximum number of threads in each socket is NC, allowing a number of configurations from 1 thread to $NS \times NC$. This definition allows us to consider configurations independently of thread positioning on the socket, that is, by considering homogeneous threads, where a thread and its siblings in a socket are equivalent. Furthermore, configurations with the same number of threads per socket but with different socket order are also considered

equivalent (e.g. AFF=$\{1,2\}$ is equivalent to AFF=$\{2,1\}$).

Finally, this definition provides a number of possible configurations in Equation 5.6 , being *NC* the number of cores per socket, and *NS* the number of sockets in the system. Considering this, the model provides the estimation for all the different $numConf$ affinities (AFF) in the system, and allows to select the configuration with the minimum estimated execution time.

$$numConf = \binom{NC+NS}{NS} - 1 \tag{5.6}$$

Estimating TCM & Execution Time

In order to estimate the TCM generated in a socket from a given affinity configuration, we represent the estimated TCM by expression 5.7.

$$estTCM(AFF, aff_s) = \frac{TCM_1}{sizeAFF(AFF)} \times aff_s \times cf_{aff_s} \tag{5.7}$$

Where s is the number of socket, and $sizeAFF(AFF)$ expresses $\sum_{x=1}^{NS} aff_x$, i.e., the total number of threads for the AFF configuration.

Finally, time estimation for the affinity configuration is given by the ideal execution and the overhead time ($TOvhd$) as shown in expression 5.8.

$$estTime(AFF) = TOvhd(AFF) + idealTime(AFF) \tag{5.8}$$

Where $TOvhd(AFF)$, presented in 5.9, is the calculated overhead depending on the data access pattern. If the pattern is unknown, the $TOvhd(AFF)$ value can be interpolated between the best and the worst case scenario. The serialized access pattern considers the worst case scenario, summation (SUM) of all the socket overhead, and on the other hand, the best case scenario is presented by the fully parallel memory access between sockets (MAX), using the maximum value overhead estimated on all sockets.

$$TOvhd(AFF) = \begin{cases} \sum_{s=1}^{NS} \quad TOvhd(AFF, aff_s), & \text{Serialized Mem. Access.} \\ \\ \max \quad TOvhd(AFF, aff_s), & \text{Parallel Mem. Access} \end{cases} \tag{5.9}$$

Therefore, in order to describe the overhead time per socket we define $TOvhd(AFF, s)$ expression 5.10 that represents the overhead generated by TCM in a socket minus $idealTCM_{aff_s}$,

which is corrected with the β value, that corresponds to its concurrency degree (aff_s) measured in a single socket. The $idealTCM_{aff_s}$ is obtained from $\frac{TCM_1}{size(AFF)} \times aff_s$

$$TOvhd(AFF, aff_s) = (estTCM(AFF, aff_s) - idealTCM_{aff_s}) \times \beta_{aff_s} \tag{5.10}$$

This model provides the execution time estimation for the AFF vector configuration, just by considering the values of a single socket execution, and can be applied for all the affinity configurations present in the system. Selecting the optimal configuration is not always trivial, but, applying the model, it is possible to provide an estimation for each configuration an select the one with minimum execution time.

5.5 Experimental validation

In this section we present the experimental validation of the proposed performance model. We have used two different multicore architectures (Table 5.2), Dell T7500 workstation and a node from LRZ SuperMUC computer (FatNode), and representative regions of interest for the memory bound applications SP (Scalar Pentadiagonal solver), and the MG (Multi-Grid) benchmarks from the NAS Parallel Benchmarks [65] NPB3.3.1-OMP, using different workloads.

Firstly, we introduce application and system characterization. Next we present the validation of the model on the T7500 system with two sockets per node and 6 cores in a socket, and the validation of the model on the FatNode system with 4 sockets and 10 cores per socket, allowing us to evaluate the model for a greater number of configurations.

By using the definition of AFF provided in the previous section, the total number of possible configurations ($numConf$) for the T7500 system is 27, and for the FatNode system is 1000.

The SP application has 4 principal parallel regions, where 3 parallel loops (at x_solve, y_solve, and z_solve functions) represent each one about 15% of the total execution time, and one parallel region (at the rhs function) with inner loops representing between 20% and 40% of the execution depending on the degree of parallelism. The MG application presents 2 parallel loop regions of interest, Reg_011 (mg.f 614-637) and Reg_013 (mg.f 543-566), representing 28% and 16% respectively of total execution time.

To compare the measurements and the estimations, we have executed them for different number of threads and representative affinities. We have used the ompP [44] profiler to obtain

Table 5.2: System hardware characteristics at node level.

Properties	**T7500**	**FatNode**
Instruction set	Intel 64	Intel 64
Processor	Westmere-EP Intel Xeon E5645 (2,4GHz)	Westmere-EX Intel Xeon E7-4870 10C (2,4GHz)
# of Sockets	2 sockets	4 sockets
#cores per socket	6 cores	10 cores
Hyperthreading	Yes (x2)	Yes (x2)
Total #PU	24 threads	80 threads
L1 cache size	32 KB (I and D)	32 KB (I and D)
L2 cache size	256 KB	256 KB
L3 cache size	12 MB unified	30 MB unified
Main Memory	96 GB	256 GB
Time per cycle	0.4166 ns	0.4166 ns
L1 cache latency	1.434 ns 4 cycles	1.673 ns 4 cycles
L2 cache latency	3.586 ns 9 cycles	4.182 ns 10 cycles
L3 cache latency	18.5 ns 45 cycles	20.172 ns 48 cycles
Local Main Mem. lat.	77 ns 185-288 cycles	116.254 ns 278 cycles

performance information at application and at parallel region level. In addition, ompP is integrated with PAPI [31] to obtain hardware counters information. We considered the full profiling information for the MG benchmark, and a reduced number of iterations for the SP benchmark, being 100 iterations for class C, and 10 iterations for class D.

Information given by PAPI is based on preset counters. We observe that the load (LD_INS), store (SR_INS), total (TOT_INS), and floating point (FP_INS) instructions are distributed evenly between threads. TCM for cache levels 1, 2 and 3 (L1_TCM, L2_TCM, and L3_TCM) have been evaluated to characterize the memory contention problem of the applications.

The execution with *likwid-pin* tool [40] allows to pin threads to cores in order to evaluate the affinity. The affinity labeled as $AFF0$ assigns threads to cores at the same processor, until it is full. Affinities $AFFi$ define a Round-Robin distribution between sockets from a list of

Table 5.3: T7500 system. Input data for x_solve parallel region from SP benchmark class C.

$sizeT(s1,s2)$	Measured Time (s)	Measured TCM	cf_i	β_i
1(1,0)	123	1.19×10^{8}	1.00	0.0
2(2,0)	63	1.46×10^{8}	1.23	1.65×10^{-10}
3(3,0)	57	7.89×10^{8}	6.61	2.58×10^{-10}
4(4,0)	70	34.4×10^{8}	28.84	1.47×10^{-10}
5(5,0)	74	59.3×10^{8}	49.69	1.09×10^{-10}
6(6,0)	78	78.9×10^{8}	66.10	0.94×10^{-10}

current threads to be executed, where *i* represents the chunk size of threads from the list to assign to each socket, and until the socket is filled. For example, in a two socket system with 6 cores per processor, execution of 9 threads with $AFF3$ assigns the first 3 threads to socket 1, next 3 threads to socket 2, and the last 3 threads to socket 1.

The *numatcl* utility has been used to evaluate the behavior for different memory mappings, by using two configurations, *localalloc* to force allocation closer the master thread, and *interleave=all*, where memory is allocated evenly between all set of NUMA nodes.

5.5.1 Applying the model for the SP application on the T7500 system.

In this section, we apply the model to a parallel region of interest to evaluate the NAS SP class C on T7500 system, in order to compare the model estimation against the execution times for two different affinity distributions.

The information from the profiled execution on a single socket is used, considering the values from 1 thread to total number cores per socket (# cores per socket. in Table 5.2).

First step is to compute the CF vector and BF vector using TCM and times per parallel region. Input data is shown on Table 5.3.

Following this, the CF is used to estimate the TCM for a specific AFF configuration. In this example, if we consider AFF1, distributing threads from 1 to total number of cores in the T7500 system, in a Round Robin distribution, we obtain the different configurations expressed in Table 5.4, shown in column ($sizeT(s1,s2)$). Applying expression 5.7 for each combination of number of threads in the sockets we obtain the $estTCM(AFF,i)$ per socket and the cumulative estimation $Cum.TCM$, which is presented in column $EstimatedTCM$. For this configuration, the relative error of the estimated TCM and measured TCM is presented in column $\%RelativeError$.

Relative error is less than 20%, and we can observe that our estimation represents the

Table 5.4: T7500 system. SP class C with affinity $AFF1$. Estimation and evaluation of TCM for parallel region x_solve.

$NunThreads$	Distribution (s1,s2)	Measured.TCM	Estimated TCM	%Relative Error
1	(1,0)	1.19×10^8	1.19×10^8	0
2	(1,1)	1.16×10^8	1.19×10^8	2.57
3	(2,1)	1.63×10^8	1.37×10^8	15.73
4	(2,2)	1.81×10^8	1.73×10^8	18.87
5	(3,2)	6.28×10^8	5.63×10^8	15.24
6	(3,3)	9.05×10^8	7.90×10^8	12.67
7	(4,3)	2.50×10^9	2.31×10^9	7.87
8	(4,4)	3.82×10^9	3.44×10^9	9.91
9	(5,4)	5.01×10^9	4.83×10^9	3.54
10	(5,5)	6.14×10^9	5.94×10^9	3.34
11	(6,5)	6.94×10^9	7.00×10^9	0.90
12	(6,6)	7.45×10^9	7.90×10^9	5.96

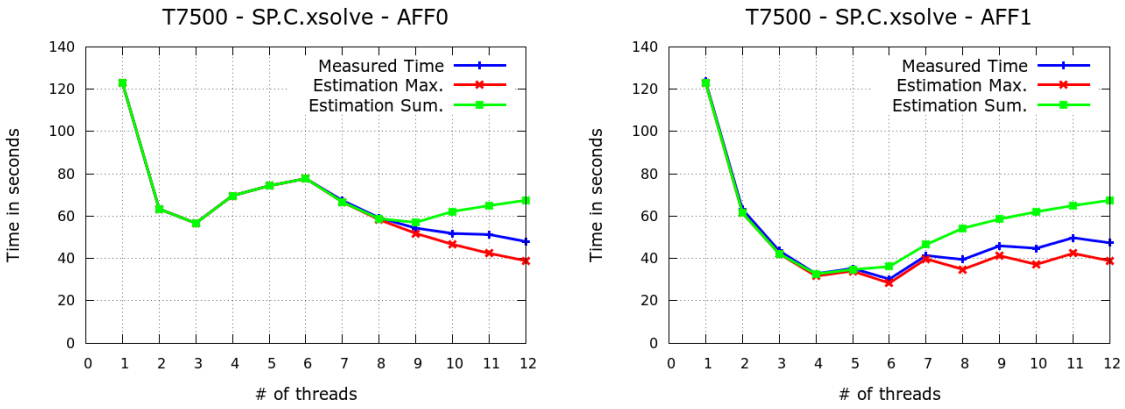
behavior of the measured values.

Using the estimated TCM, we apply expression 5.8 in order to obtain the final estimation time ($estTime(AFF1)$) for the affinity 1. For this case, we evaluate two different estimations, one by considering a serialized memory access and a second one that assumes an ideal parallel memory access. Therefore, the first case considers the overhead as the summation of overhead times per socket, and the second assumes full parallelism on memory accesses, implying that the overhead time is generated by the slowest socket, therefore by the maximum time estimation of sockets.

Both estimations are shown for the two affinity distributions (0 and 1) presented in Figure5.2, which shows that the measured time is in between the two estimated boundaries, and in this case is similar to $EstimationMax.$, meaning that the memory accesses are parallelized between the sockets. Furthermore, the $EstimationMax.$ presents the same behavior and lead us to identify the best configuration, which in this case is the $AFF1$ using 6 threads (equivalent to socket configuration $\{3,3\}$), and median error for the best estimation is 5%, and the average error is less than 8%.

5.5.2 Selecting a configuration for SP and MG benchmarks on FatNode

We present the application of the model for SP and MG, with different workloads, on the FatNode system.



(a) Evaluation for AFF0 (b) Evaluation for AFF1

Figure 5.2: Evaluation of execution time between estimated boundaries.

The experiments are configured to evaluate the two boundaries at memory level. We use the *numactl* tool to allocate memory near to master thread (localalloc), to achieve a serialized memory access at socket level, and interleaved allocation (interleave=all) to force data distribution between sockets and parallel memory accesses.

The model is applied considering the single socket measurements and the results are shown in Table 5.5.

Table 5.5: Selection of configuration for SP and MG benchmarks

System	Bench.	Par.Reg.	Best Conf. Measured	Best Conf. Modeled	%Avg Error	Mem.Model
FatNode	SP.C distr.	x_solve	AFF1(24) = {6,6,6,6}	AFF1(32) = {8,8,8,8}	4.64	MAX
	SP.C loc.	x_solve	AFF1(20) = {5,5,5,5}	AFF1(20) = {5,5,5,5}	8.55	SUM
	SP.D loc.	x_solve	AFF1(9) = {3,2,2,2}	AFF1(4) = {1,1,1,1}	11.40	SUM
	MG.C loc.	R0011	AFF1(32) = {8,8,8,8}	AFF1(40) = {10,10,10,10}	13.18	MAX
	MG.C loc.	R0013	AFF1(32) = {8,8,8,8}	AFF1(40) = {10,10,10,10}	21.50	MAX

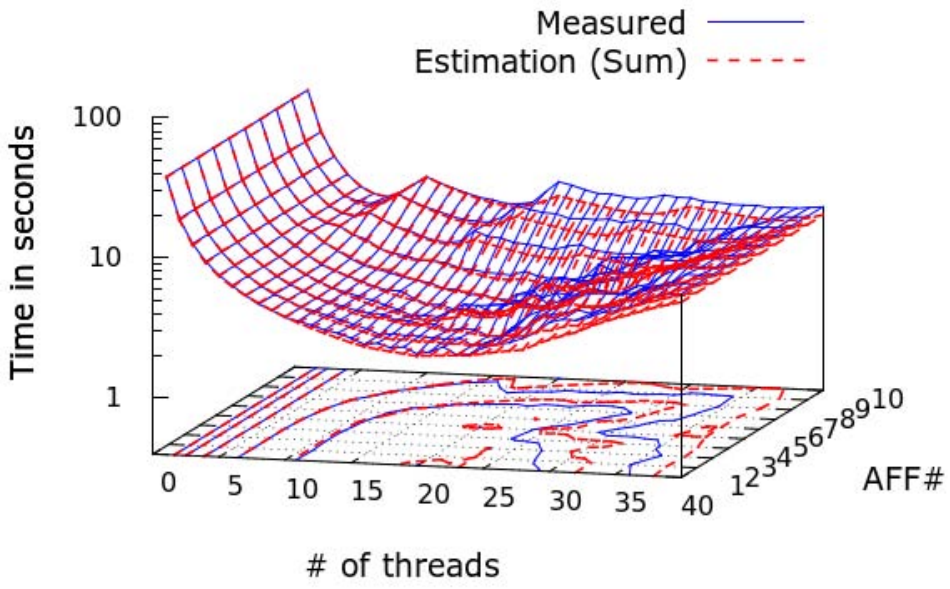


Figure 5.3: SP_C_xsolve local allocation

We can observe on Table 5.5 for SP benchmark that local allocation provides a serialized memory access. This is because data needs to be accessed through the same socket, and this contention provides a serialized behavior. For the distributed allocation, the memory access pattern allows more parallelism, improving performance and minimizing the memory bottleneck. The model has provided a configuration with minimum execution time and an average error of less than 14%.

MG has been forced with local allocation, however, it uses a different data access pattern and higher workload. We have observed that memory access is not fully parallelized neither serialized, therefore we used the closer boundary *Max.Estimation*, which not represents exactly the data access pattern increasing the error.

5.5.3 Exploration of the affinity configurations.

In this section we discuss the benefits of applying the model in a system with multiple sockets, and the speedup achieved by allowing the selection of a configuration with the model compared to the execution with all threads.

The main point is to rapidly detect memory bottlenecks in parallel regions, and select a configuration that minimizes the contention overhead. Also, to provide an estimation approach for all the configuration ranges without a full execution.

We present a model that provides an estimation for all the configuration ranges, which can

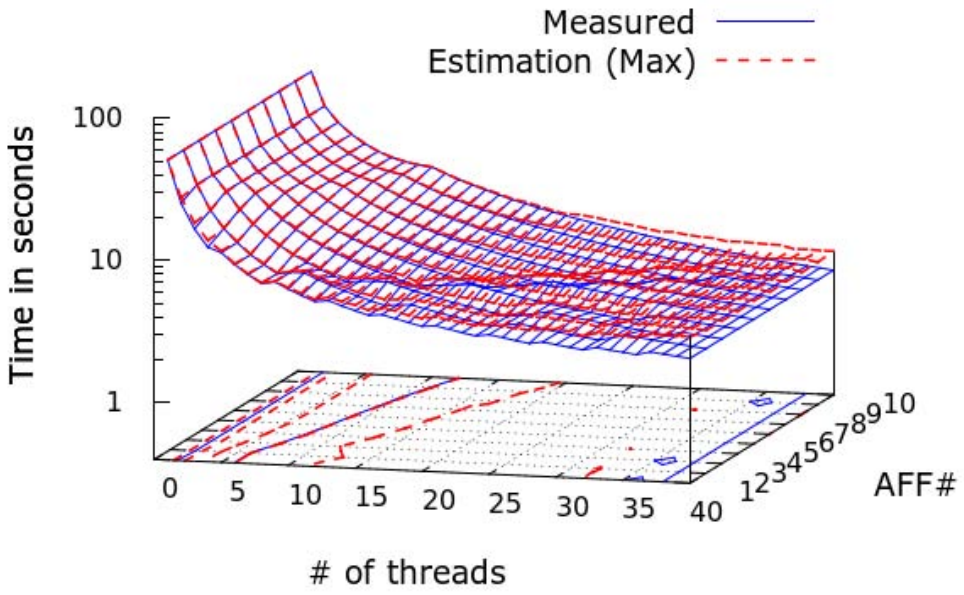


Figure 5.4: MG_C_R0011 local allocation

be applied with a minimum characterization on a single socket. Figures 5.3 and 5.4 show a subset of 10 configuration affinities (considering the definition in 5.5) for the FatNode system.

Figures 5.3 and 5.4 show the measured times and the estimated execution times. We can observe that Figure 5.3 presents a memory contention problem when using a full thread execution. The minimum for measured and estimated execution times is shown on a contour surface. The minimum execution time is achieved by using about 20 threads on the configuration that provides less concurrency per socket (e.g. AFF1(20)= $\{5,5,5,5\}$, that is, using half threads per socket).

Figure 5.4 shows that MG does not present significant variation between affinities, and time is reduced using more threads.

Finally, we present in Table 5.6 the comparison between an unguided execution using all threads, and the configuration provided by the model. The speedup is calculated using the measured time for full execution and measured time for the selected configuration.

Even though the ideal configuration is not detected for all cases, the selection has provided a configuration with a maximum speedup of 2.74, for the SP class C, with an affinity 1 with 20 threads. Moreover, the minimum speedup is 1, meaning that the application does not shows memory contention, neither benefiting from reducing the number of threads or modifying the affinity.

Table 5.6: Execution time for selected configuration and speedups.

Bench.	**Max threads Conf.**		**Selected Conf.**		**Speedup**
	Threads per socket	Measured Time (s)	Threads per socket	Measured Time (s)	
SP.C.xsolve distr.	{10,10,10,10}	3.65	AFF1(20) = {5,5,5,5}	2.57	1.42
SP.C.xsolve.loc.	{10,10,10,10}	6.81	AFF1(20) = {5,5,5,5}	2.49	2.74
SP.D.xsolve loc.	{10,10,10,10}	23.58	AFF1(4) = {1,1,1,1}	20.27	1.16
MG.C.R0011 loc.	{10,10,10,10}	4.09	AFF1(40) = {10,10,10,10}	4.09	1.00
MG.C.R0013 loc.	{10,10,10,10}	2.26	AFF1(40) = {10,10,10,10}	2.26	1.00

5.6 Summary

We have presented a performance model to estimate the LLC misses and to estimate the execution time based on an execution of a small set of configurations. This model allows to estimate any possible configuration of affinity and number of threads for the system. The performance model has been applied for the NAS SP and MG applications for classes C and D in two different architectures. The results show an average time error of less than 14%. Despite the error, the time estimation preserves the measured behavior that lead us to select automatically a configuration, and the possibility to improve performance compared with the default configuration.

Our model can rapidly detect memory bottlenecks on each parallel region in an application, and it is possible to identify a configuration that minimizes the contention overhead.

This model has the drawback that it provides two performance boundaries (*Max* and *Sum*) in the cases where the memory access pattern of an application is not completely serialized or parallel. Furthermore, when the boundaries are widely separated, and the measured time is in between, the error increases.

In order to provide a more accurate estimation, we have notice that it is needed to consider the overheads of accessing data between different sockets.

Finally, it would be desirable to iterate upon the methodology in order to define a model that considers the application pattern access. We assume that by knowing the pattern access of the application will allow us to identify which one of the two boundaries is closest to the real behaviour.

6

Performance model based on profiling the memory footprint

"L'interminable est la spécialité des indécis."

De l'inconvénient d'être né - **Emil Cioran**

In this chapter, we present a runtime performance model based on the pre-characterization for small workloads and a dynamic tuning strategy to estimate a configuration of number of threads and thread distribution to improve performance by detecting and avoiding memory contention.

6.1 Introduction

In this chapter, we propose a performance model to estimate at runtime the execution time and the ideal configuration by defining a characterization of the memory footprint for small workloads to identify the degree of parallelism which can lead to a performance degradation. In the previous chapter, the model required a dynamic exhaustive characterization in a single processor execution by analyzing all the thread configuration combinations, which at the end, limits the scalability of the model by increasing the overhead at runtime.

To minimize memory contention, it is possible to analyze the application access pattern for a concurrent execution to determine per thread memory characteristics, such as required memory, total number of accesses, reutilization degree, etc. On an OpenMP parallel loop, every thread executes iterations or units of parallel work, and each unit has an associated memory footprint. This active data usage can be characterized for many applications for a set of workloads. By doing this, it is possible to expose a latent performance problem given a different workload and a thread configurations and, in most cases, to provide a configuration that minimizes the performance contention. To do that, we analyse the reutilization degree of this data.

This chapter is sctructed as follows. Section 6.2 defines the objective and contributions of the model. Section 6.3 presents related work on performance models and memory access pattern analysis. The methodology and the performance model is defined in Section 6.4, and the validation through experimentation of two applications in Section 6.5, showing a maximum speedup of 2.5x. Finally, a summary of this chapter is shown in Section 6.6.

6.2 Objective

The objective of the current model is based on analyzing the memory access pattern of the application in order to obtain characteristics which can lead to identify potential performance factors within the application.

To do that, we have developed a tracing tool in order to obtain information about the memory access pattern of the application. By doing this, the overheads of generating the trace increase dramatically, both in the volume of gathered information and trace generation time. However, the methodology considers to perform an off-line characterization to extract such characteristics from executions using small workloads. Further, this information can be extrapolated for real workloads at runtime.

Identically as the previous model, we assume the following conditions for the context;

1. The application is iterative and all the iterations have a uniform workload;
2. Workload is evenly distributed among threads;
3. Performance degradation is mainly due to memory contention at the main memory, generated by the application memory access pattern at LLC;
4. All the cores in the processor are homogeneous

Taking into consideration these conditions, contributions from this model are the following:

- A methodology, based on memory trace analysis, to extract the memory footprint of parallel loop regions from an OpenMP application in order to identify memory intensive parallel regions.
- Estimation of critical configurations of the application for a specific hardware configuration.
- An execution time estimation for a configuration of a parallel region to dynamically tune the application by identifying a configuration that minimizes memory contention.

6.3 Related work

Performance of parallel applications is expected to increase proportionally to the number of used resources. We have seen in chapters 3 that there are several factors that limit their scalability. On multicore architectures, the access to the memory hierarchy is possibly the most important limiting factor, specially for memory intensive applications [7]. This behaviour can be noticed significantly on the LLC (Last Level Cache) in current multicore processors, mainly because the increase on cache evictions to main memory reduces the overall data cache reuse at this level.

During a parallel program execution, its aggregated working dataset can be defined as the *memory footprint* [79] [80] [81] [82]. The continuously increasing integration of number of cores accessing shared cache levels can lead to a performance degradation [83]. One of

the reasons is an imbalanced relation between the per core cache size and the per thread application footprint [84].

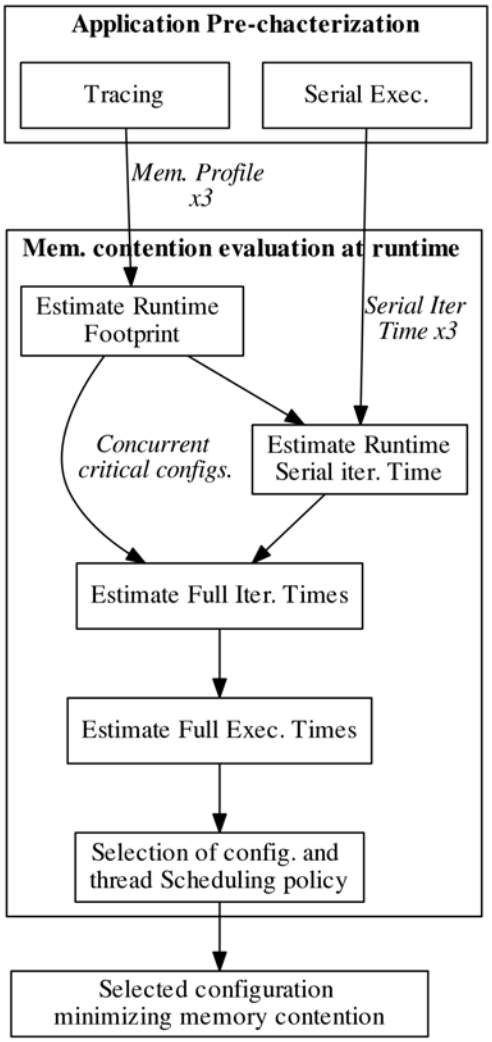


Figure 6.1: Methodology to select a configuration that avoid memory contention based on the analysis of the concurrent memory footprint in LLC

6.4 Methodology for selecting a configuration to avoid memory contention

The memory contention problem can be generated by a large number of requests to main memory. In parallel applications, this problem can be increased at certain levels of concurrency because of the inter-relation of a shared resource utilization and the application data access

patterns. In this model, we evaluate the average occupation of LLC, and the application data reutilization patterns.

To obtain the memory footprint of an application, we have developed a tracing tool, in a form of a *pintool*, using Intel's PIN framework for dynamic instrumentation [8], which extracts the memory trace in each parallel region iteration.

Figure 6.1 defines the methodology to select a configuration that avoid memory contention based on the analysis of the concurrent memory footprint in LLC. We use the application's characteristics to identify memory contention at runtime which, at the end, allows us to dynamically apply a strategy to tune the number of threads and thread distribution at runtime. With this aim, we provide a way of identifying performance degradation due to memory contention. Furthermore, because the pre-characterization is performed off-line, the overheads at runtime are negligible.

6.4.1 Trace generation and characterization of iteration footprint

We instrument the application using the *pintool* to obtain a memory trace in the format described in Equation 6.1. The trace is processed to obtain the memory footprint of the most reaccessed data. To do this, consecutive addresses of memory are combined, and the most significant regions are used to compute the memory footprint. Consecutive regions smaller than a cache line and reutilization degrees smaller than a factor of 4 are discarded, meaning that for the selected memory ranges, bigger than a cache line, there is an average of more than 4 accesses per each element. This criteria is defined in order to discard effects such as reutilization that can be solved at low level caches, or with no significant impact in performance.

Following this, we describe the trace and profiled information obtained from an individual iteration.

- The minimal information unit obtained from the *pintool* is the *memOp*. Every memory operation is described by *event_id*, as the unique key identifier on the temporal access pattern, *type* refers to the memory operation (load or store), the actual virtual memory *address* and the *data_size* of the element accessed.

$$\text{memOp} = \{\text{event\_id}, \text{type}, \text{address}, \text{data\_size}\} \tag{6.1}$$

- We extract the spatial access pattern by sorting the memory trace and grouping consecutive memory references. Consecutive *elements* are combined into *streams*

of memory references with equal or less than *data_size* (e.g. 8 Bytes) spacing between its elements. The *stream* information (eq. 6.2) is composed by the different number of *elements* within the range, and the total number of references into the address space (*accesses*). The multiset representation of streams is defined in 6.3 to group streams with same properties. Memory references from the trace, that are not aligned into sequential accesses, are discarded.

$$\text{stream} = \{\text{elements}, \text{accesses}\} \tag{6.2}$$

$$\text{Streams} = \cup\{\text{stream}, \#streams\} \tag{6.3}$$

- Streams information is extended within cluster in Eq. 6.4. We extract metrics *reutil* (eq. 6.6) as the reutilization degree of the elements described by the cluster, and *clusterfootprint* (eq. 6.7), which is the total size of the streams. The union of clusters is defined as a set in Eq. 6.5.

$$\text{cluster} = \{\text{elements}, \text{accesses}, \#\text{streams}, \text{reutil}, \text{cl\_footprint}\} \tag{6.4}$$

$$\text{Clusters} = \cup\text{cluster} \tag{6.5}$$

$$\text{reutil} = \text{accesses}/\text{elements} \tag{6.6}$$

$$\text{cl\_footprint} = \text{elements} \times \#\text{streams} \times \text{data\_size} \tag{6.7}$$

- The iteration footprint (eq. 6.8) is obtained from the aggregation of all the iteration clusters footprints. The footprint makes reference to memory elements with high reaccesses degree along the iteration execution, which must be preserved into LLC to improve locality. The ratio of the memory footprint regarding the total memory operations in the parallel region trace (eq. 6.9).

$$\text{footprint} = \sum_{i=1}^{\#\text{clusters}} \text{cl\_footprint}_\text{i} \tag{6.8}$$

$$\text{footprint\_ratio} = \text{footprint}/\#\text{memOp} \tag{6.9}$$

6.4.2 Estimation of memory footprint at runtime

The memory trace generates a huge amount of information, in the order of GBytes for a few execution seconds. Therefore, we obtain a memory footprint from executions of the application using small workloads. Afterwards, we estimate the memory footprint from a runtime execution for a real workload. To infer this value, it is possible to use runtime information provided by hardware counters, and also by the runtime library.

First, it is necessary to assume that the information on the total number of memory references obtained at runtime through hardware counters is going to be equivalent to the information given by the *pintool* trace.

Second, as the memory trace is obtained from a single iteration of the parallel region, the runtime information must be normalized as shown in eq.6.10. We use hardware counters to count the number of loads and stores. The number of iterations is provided by the OpenMP runtime library, which is in charge of scheduling iterations (work) among threads. This is done by a wrapper of the OpenMP runtime library which intercept OpenMP calls.

$$\text{rt\_tot\_memOp} = (\text{PAPI\_LD\_INS} + \text{PAPI\_SR\_INS})/\#\text{iters} \tag{6.10}$$

Third, we obtain the memory footprint per iteration by applying the representative ratio (eq.6.9) from the traced workloads to the rt_tot_memOp (eq. 6.10).

$$\text{rt\_footprint} = \text{rt\_tot\_memOp} \times \text{footprint\_ratio} \tag{6.11}$$

6.4.3 Estimation of execution time for all configurations at runtime

After the characterization of small workloads to determine the parallel regions memory footprint, we obtain the current execution time for the first execution of the parallel region with the maximum concurrency ($maxPar\_iterTime$ on *maxCores* configuration). The total parallel region execution time is normalized to express iteration execution time.

Given that a loop iteration is the minimum work unit in an OpenMP parallel loop, the iteration execution time should be constant for any thread configuration in the absence of memory contention.

We consider the iteration execution time as the minimal execution unit, and this is because parallel loops in OpenMP express an SPMD paradigm. Therefore, the computation time for this unit should be constant, and that holds true while there is no memory contention.

By using the $maxPar\_iterTime$ and estimating the ideal execution time, we propose an

heuristic to determine the execution time for all the configurations of threads in a multicore multisocket system. Additionally, this heuristic allows to estimate execution times for different thread distribution policies.

First, to estimate the sequential execution time for the current workload, we use the reference values obtained along the characterization of small workloads. We use serial execution times obtained from at least three traced workloads (Eq. 6.12 and 6.13). We assume a high correlation coefficient between memory footprint and sequential execution time. To do that, we obtain the coefficients of a linear regression function from the three characterized small workload considering the correlation of their sequential execution time and their memory footprint, described as function (eq.6.14). Using the lineal regression function, we interpolate the sequential execution time for the current footprint ($rt\_footprint$).

$$\text{sample\_times} = \{\text{execTime\_wkld\_1}, \text{execTime\_wkld\_2}, \text{execTime\_wkld\_3}\}; \tag{6.12}$$

$$\text{sample\_footprint} = \{\text{footprint\_wkld\_1}, \text{footprint\_wkld\_2}, \text{footprint\_wkld\_3}\}; \tag{6.13}$$

$$\text{ideal\_iterationTime} = F(\text{sample\_times}, \text{sample\_footprint}, \text{rt\_footprint}); \tag{6.14}$$

Second, we identify the iteration time (eq.6.16) for a number of threads configuration regarding its occupation on the LLC. To do this, we determine the level of contention at the LLC (eq.6.15). We assume the iteration time is constant for configurations with no contention, and iteration time is going to increase up to $maxPar\_iterTime$ for configurations with contention, starting from the first configuration of number of threads that overfills the LLC ($first\_contention$, eq.6.17). As described in algorithm 2 we use a linear regression function to interpolate between the last constant iteration time point and the maximum iteration time point (provided at runtime).

$$\text{concurrent\_memory} = \text{footprint} \times \text{num\_threads} \tag{6.15}$$

$$\text{iterationTime} = \begin{cases} \text{constant} & \text{concurrent\_memory} < \text{cacheSize} \\ G(\text{last\_constant}, \text{max\_iterTime}) & c\text{oncurrent\_memory} \geq \text{cacheSize} \end{cases} \tag{6.16}$$

$$\text{first\_contention} = min(\text{nt}) \ ; \ \textit{where} \ \text{concurrent\_memory} \geq \text{cacheSize} \tag{6.17}$$

We use the estimation of $iterationTime$ for a given thread $distribution\ policy$, which is the description of how threads are binded to processors for the current system. For example, a

compact distribution places threads in closer cores, while a scattered policy distributes them as evenly as possible across the entire system.

Algorithm 2 shows the steps to estimate all the iteration times for a given distribution policy.

Algorithm 2 Estimation of iteration time for all possible number of threads using a particular scheduling policy. *sched(x)* is a function that returns the last assigned processor id., and interpolate() is a function for a linear regression interpolation.

Data: nt=2

Result: iterationTime[maxCores]

```
1  iterTime[1] = ideal_iterTime
2  final_point = {maxPar_iterTime, maxCores}
3  while nt ≤ maxCores do
4      proc_id = sched(nt)
5      footprint = base_footprint * get_threads_on_proc (proc_id)
6      if footprint ≤ cacheSize then
7          iterationTime[nt] = iterTime[nt-1]
8          base_point = {iterTime[nt], nt}
       else
9          iterationTime[nt] = interpolate( nt, base_point, final_point )
       end
10     increase nt ; increase_threads_on_proc(proc_id)
   end
```

Finally, we determine the final execution time (eq. 6.19) for all thread configurations by transforming iteration times into parallel region execution times. To do this, we apply eq. 6.18 to determine the maximum number of iterations scheduled by the default policy in an OpenMP parallel loop (static), and use this value to multiply the iteration time for a given configuration.

$$\text{sched\_iters}_{(\text{nt})} = RoundUp(\text{num\_iters}/\text{nt}) \tag{6.18}$$

$$\text{ExectionTime}_{(\text{nt})} = \text{sched\_iters}_{(\text{nt})} \times \text{iterationTime}_{(\text{nt})} \tag{6.19}$$

The estimation for different distribution policies can be used in a multicore multisocket system to identify the configuration that minimizes the execution time.

6.5 Experimental Validation

In this section, we apply the proposed methodology on two applications, SP (Solver Pentadiagonal) from NAS Benchmark suite, and a version of Stream Benchmark. Both benchmarks have been selected because they are memory intensive applications which suffer from performance degradation due to memory contention at certain degree of parallelism. This has been previously proven for SP benchmark showing a benefit from tuning their configuration of number of threads and thread binding in the previous evaluation. The stream benchmark has been modified in order to meet the assumptions in section 6.2.

The benchmarks have been evaluated on the hardware architectures described in Table 6.1. The systems are compute nodes from Marenostrum supercomputer at BSC (Barcelona Supercomputing Center) and SuperMUC supercomputer from LRZ (Leibniz Supercomputing Center).

Table 6.1: Description of system architectures for performance evaluation

	MN3 node (BSC)	Fat Node (LRZ)	Thin Node (LRZ)
Proc. Name	Xeon E5-2670	Xeon E7-4870	Xeon E5-2680
Family	Sandy Bridge-EP	Westmere-EX	Sandy Bridge-EP
Frequency	2.6GHz	2.4GHz	2.7GHz
Sockets per node	2	4	2
LLC size per socket	20MB	30MB	20MB
System Memory	32GB	256GB	32GB

6.5.1 Obtaining memory footprints and execution times

To apply the performance model at runtime, the characterization of the memory footprint and the sequential iteration time for small workloads are required for both applications.

On the one hand, SP generates a complex memory access pattern. To obtain the memory footprint, we have traced the application using small workloads to obtain the spatial access pattern. The workloads are defined as classes and are associated to a problem size. Workloads starts at S (small), continues with W (workstation), and follows with A, B, and C (standard test problems), and finally classes D, E, and F for large test problems.

First, the benchmarks have been executed with classes S, W, and A, using the *pintool* to obtain the memory trace. A second execution with no instrumentation is required to obtain the iteration execution time. In both cases, the sequential version for each class is executed.

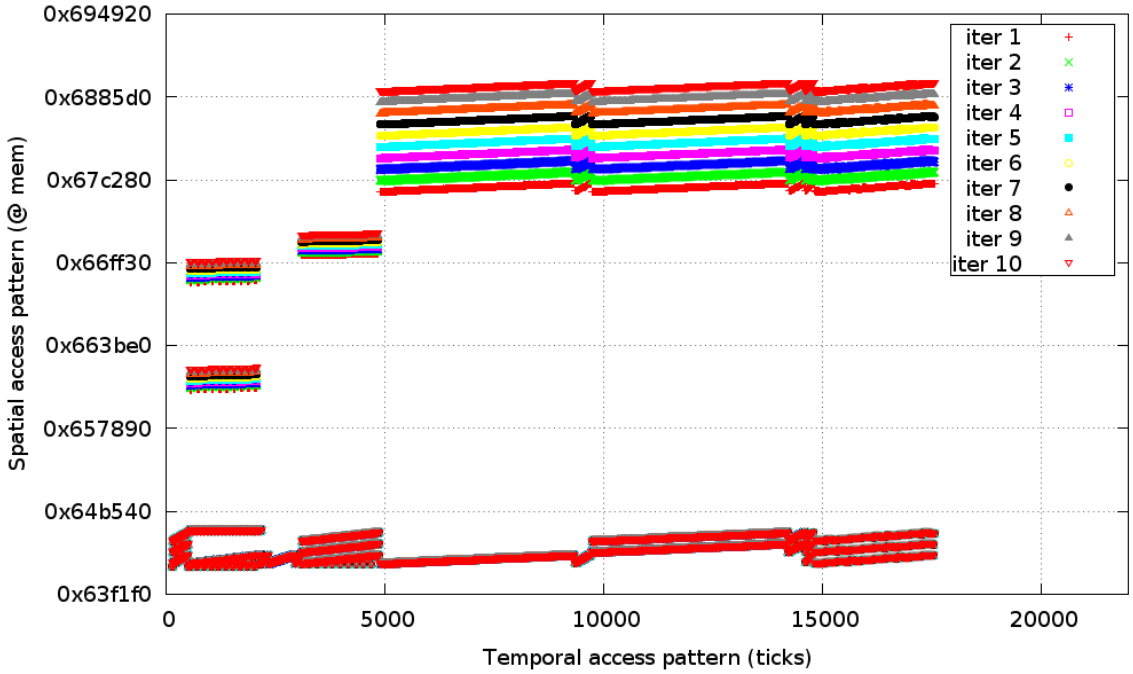


Figure 6.2: Full memory trace visualization for x_solve parallel region on SP Class W

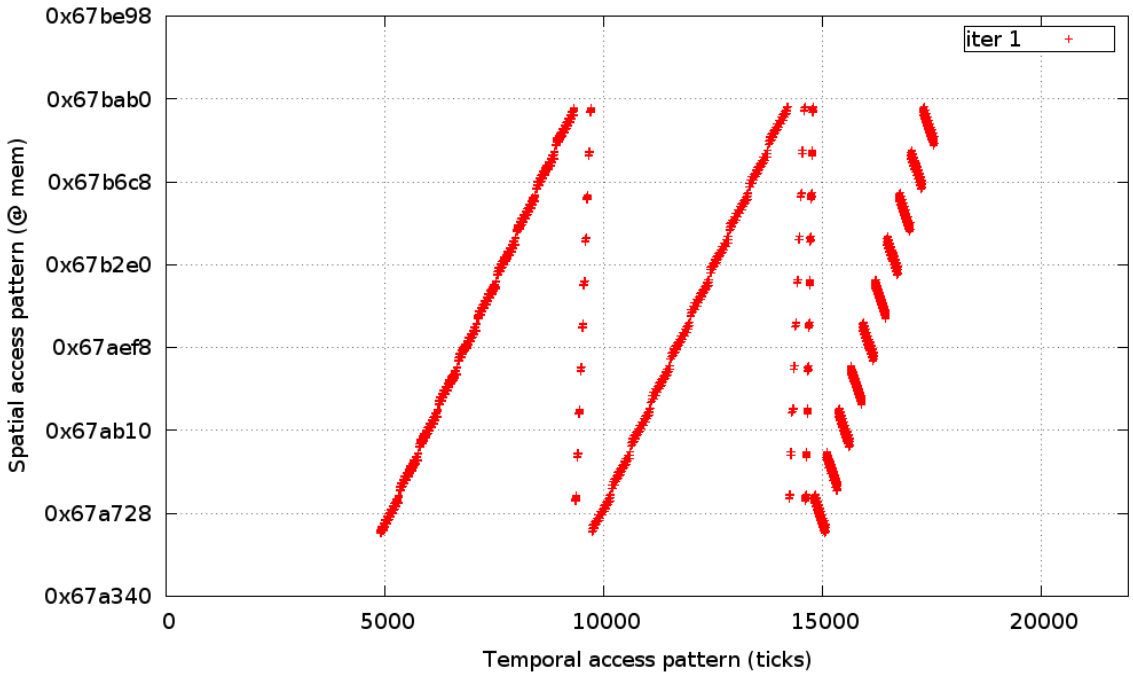


Figure 6.3: Detailed view of memory trace for x_solve parallel region on SP Class W

Figure 6.2 provides a visualization of the memory access pattern for the class W, which performs 10 iterations. Each iteration presents a reutilization pattern as illustrated on 6.3, where the x-axis represents the temporal access pattern and the y-axis represents the spatial access pattern. The footprint is evaluated on the spatial access pattern along the full iteration execution, considering blocks of memory with high degree of reutilization in the temporal access pattern.

Traces have been processed in the following way to obtain the memory footprint. We start analyzing the spatial memory pattern by identifying consecutive memory address references or *streams*, which afterwards, are combined into clusters of streams with same properties of size and elements. For SP benchmark, the clustered information within the profile represents 87% of the total memory operations of the iterations for class S, and 92% for classes W and A.

Finally, Table 6.2 shows the memory footprint (eq. 6.8) computed as the summation of sizes for every cluster of those with significant reutilization degree, in this case, and average of more than 4 references per memory element.

On the other hand, Stream Benchmark is a synthetic benchmark used to analyze memory bandwidth by the execution of four parallel regions performing a repetitive set of point to point operations in a vector structure. We use a version of the benchmark accessing matrix structures of 64 rows per 320K elements as described in Table 6.3. The benchmark is configured to repeat every vector operation 16 times, and parallelism is expressed in the outer loop of the matrix data access. That is, one iteration is going to perform 320K operations 16 times. In this case, the memory access pattern is know from the beginning, so we have analytically defined the memory footprint and validated it through hardware counters.

The footprints for the x_solve parallel region from SP Benchmark, and copy and add parallel regions from Stream Benchmark estimated for a concurrent execution in the experimental systems is described in Table 6.4. The marks show the $first\_contention$ configuration per processor on the validation system architectures. Figure 6.4 visualizes the concurrent occupancy with a threadhold set on a 20MB. In this case, Class C overfills the limit on the 6 threads configuration and Class D by only using 1 thread.

Once we have defined the memory footprint for the characterized workloads, we calculate the coefficients for a linear regression of the relation between the memory footprint and the execution time of the serial execution, in order to interpolate the ideal execution time for a new workload.

Table 6.2: Preprocessed information from traces of small classes. This information is required to estimate performance at runtime. The profile information of the memory intensive parallel region x_solve specifies the per cluster memory footprint for one iteration. Besides, inputs for the performance model such as the cumulative iteration footprint ($iterFootprint$) and the iteration serial execution time, are shown.

	cluster	accesses	elems.	#streams	stream size	reutil	clust.Size
SP.S	type 1	424	60	10	480	7.06	4,800B
	type 2	257	60	20	480	4.28	9,600B
	type 3	606	60	10	480	10,1	4,800B
iterFootprint							19,200B
seq.iterTime							5.66 s
SP.W	type 1	1,408	180	34	1,440	7.82	48,960B
	type 2	809	180	68	1,440	4.49	97,920B
	type 3	1,950	180	34	1,440	10.83	48,960B
iterFootprint							195,840B
seq.iterTime							67.72 s
SP.A	type 1	2,556	320	62	2,560	7.98	158,720B
	type 2	1,453	320	124	2,560	4.54	317,440B
	type 3	3,518	320	62	2,560	10.99	158,720B
iterFootprint							634,880B
seq.iterTime							240.29 s

Table 6.3: Stream Benchmark configuration, and the iteration footprint estimation per operation

N (vector elements)	320.000
Z (matrix rows)	64
D (Data Size) Bytes	8
ArraySize/ArrayFootprint	2,44MB
MatrixSize	156,25MB
Reaccess(repetitions)	16
copy (c=a)	4,88 MB
scalar (b=scalar*c)	4,88 MB
add (c=a+b)	7,32 MB
triad(a=b+scalar*c)	7,32 MB

Table 6.4: Estimation of memory footprint for a concurrent execution of x_solve parallel region, where $^{\dagger}first\_contention$ on MN3 and Thin nodes, and $^{*}first\_contention$ for system Fat node

Benchmark	**SP**		**Stream**	
Parallel Region	**x_solve**		**copy**	**add**
Workload	Class C	Class D	N=320K elements	
1 thread	4MB	$^{\dagger *}$**25.3MB**	4.88MB	7.32MB
2 threads	8MB	50.59MB	9.77MB	14.65MB
3 threads	12MB	75.89MB	14.65MB	†**21.97MB**
4 threads	16MB	101.19MB	19.53MB	29.30MB
5 threads	20MB	126.49MB	†**24,41MB**	* **36.62MB**
6 threads	†**24MB**	151.79MB	29.30MB	43.95MB
7 threads	28MB	177.09MB	***34.18MB**	51.27MB
8 threads	* **32MB**	101.39MB	39.06MB	58.59MB
...	...	...	...	...

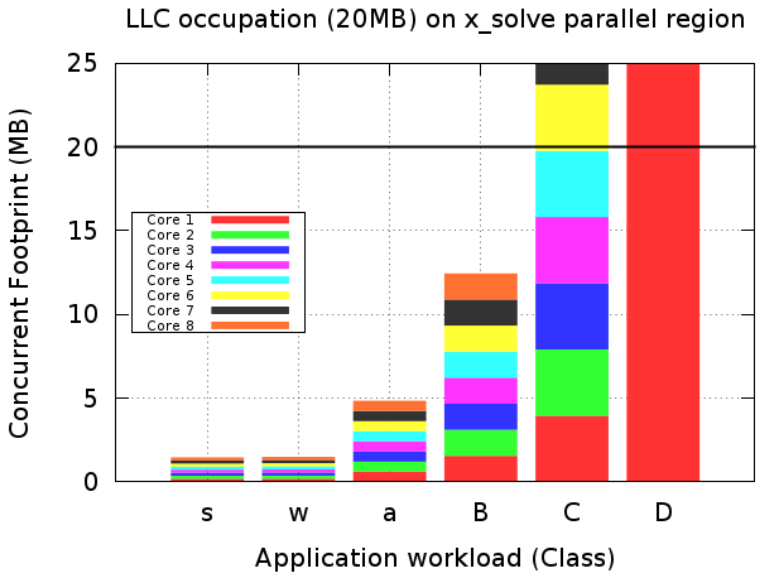


Figure 6.4: Analysis of concurrent footprint for x_solve parallel region for all workload classes in an architecture with LLC of 20MB

6.5.2 Estimating the best configuration at runtime

We proceed to execute the applications for a new workload using the maximum number of threads. The application is monitored at runtime to evaluate its performance. We assume the application to be iterative, and the analysis is done after the first iteration.

Firstly, we obtain memory operations rt_tot_memOp (eq.6.10) from hardware counters (PAPI_LD_INS and PAPI_SR_INS). We apply the maximum $footprint\_ratio$ (eq. 6.9) from the characterized workloads to deduce the current $rt\_footprint$.

Secondly, we estimate the serial execution time for the current workload. We use a linear regression of the characterized footprint with the serial execution time. Then, this function is used to interpolate the current footprint and obtain the sequential iteration time estimation.

Figure 6.5 shows a comparison of the measured times against the estimated serial execution based on the linear regression of the correletation of workload and serial execution time. As it can be seen in Table 6.5, we estimate the serial execution time with good accuracy (less than 5% error) except for class S and D. The relative error on class S is about 42% because its execution time is very small, but its absolute error is less than 3 seconds. For class D, the error estimation is generated by its memory footprint, which overpasses the LLC size limit and is expressing contention even with one thread.

Table 6.5: Estimation of serial iteration time in seconds on MN3 with 20MB LLC. The highlighted cells refer to information obtained on the characterization phase. Serial Estimation time (Est.) is obtained from an interpolation of a linear regression function between footprints and measured times for classes S, W, and A

Class	Footprint	Estimation (s)	Measured (s)	Relative Error
S	19KB	3.31	5.66	41.56%
W	191KB	71.02	67.72	4.88%
A	620KB	239.34	240.29	0.39%
B	1,638 KB	639.14	633.39	0.91%
C	4,096 KB	1,605.93	1,602.95	0.06%
D	25,907 KB	10,166.47	11,945.82	14.90%

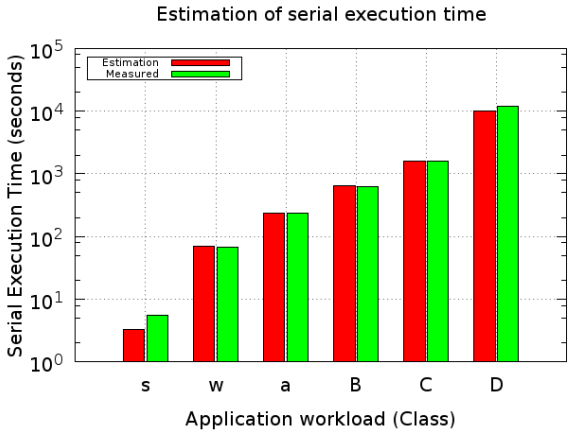
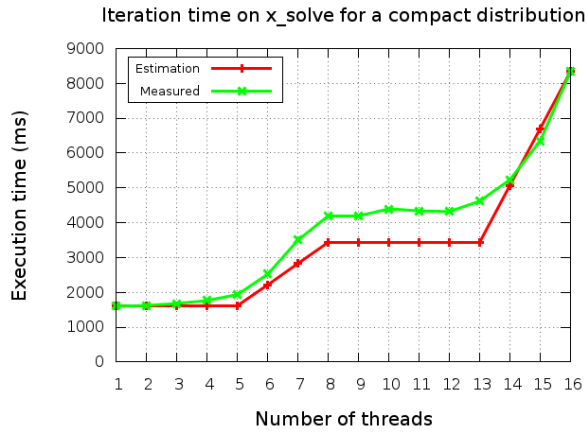
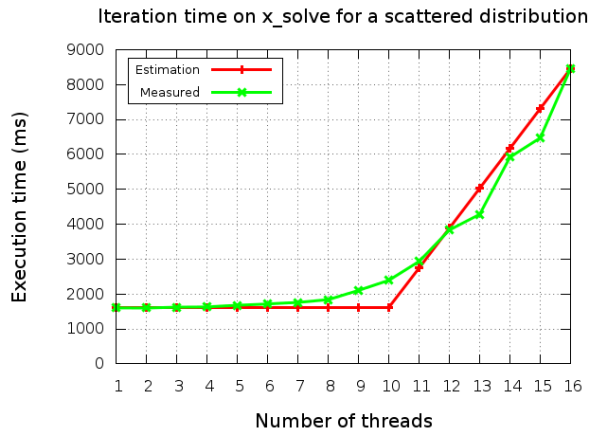


Figure 6.5: Comparison of measured and the estimated sequential iteration time using a linear regression interpolation from classes S, W and A.



(a) Compact distribution (AFF0)



(b) Scattered distribution (AFF1)

Figure 6.6: Estimation of iteration time degradation on x_solve parallel region on SP.C

We obtain an iteration times by applying the Algorithm 2 using $first\_contention$ values as identified in Table 6.4. Figures in 6.6 shows a comparison on estimated iteration times for compact 6.6(a) ($AFF0$) and scattered 6.6(b)($AFF1$) distributions in MN system.

These values are used to obtain the final estimation by applying Eq.6.19 to the iteration time estimations. The number of iterations is calculated with Eq.6.18.

6.5.3 Experimental Results

This section reports the experimental results for the x_solve parallel region from SP benchmark Class C, and the parallel regions Add and Copy from the stream bencharmk. The results are provided fro the three MN, $ThinNode$, and $FatNode$ systems.

It is important to notice that these applications express memory contention, but when the model is applied in an execution which is not expressing memory contention it provides the configuration with maximum parallelism.

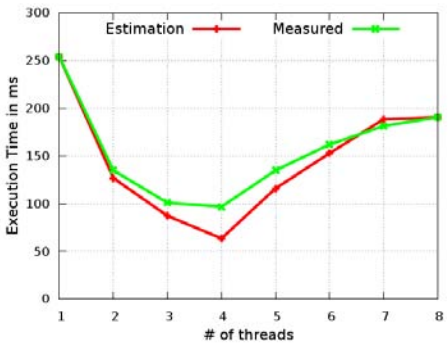
Stream benchmark is evaluated on a single processor, and the results are shown Figure 6.7, showing the estimated execution times and the measured execution times.

Figure 6.8 shows the results of the execution time estimations compared with the measured execution times for the SP application. This evaluation considers two different thread distribution policies (compact and scattered)

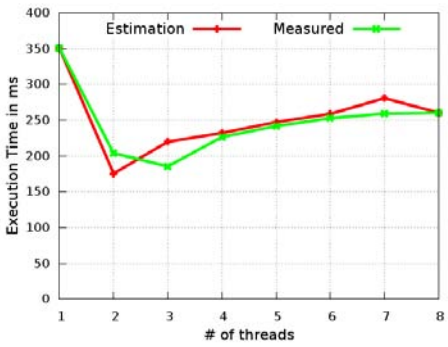
The estimations for Stream Benchmark have detected the problem of memory contention in a single processor, and the execution times estimation provides a similar behaviour compared to the real application. On these experiments, when selecting the configuration with estimated minimum execution time, and compared with the minimum execution times for the real execution, the selected configuration differs at most by 1 thread from the best configuration. By selecting the model's provided configuration instead of the maximum threads configuration is it possible to obtain a maximum speedup of 2.5x.

The different explorations of thread distribution policies in a multisocket environment on the SP benchmark shows that, with the scattered distribution on 2 processors an improvement of 2.30x speedup can be achieved using 8 threads (4 threads per processor).

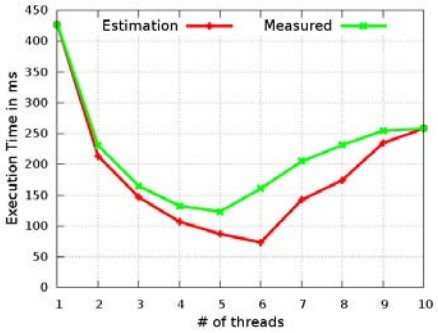
Finally, Table 6.6 shows the summarized comparison of speedups obtained with the model's provided configurations and compared with an ideal configuration selection.



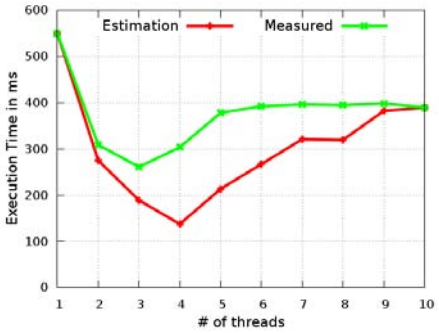
(a) MN3 – Copy



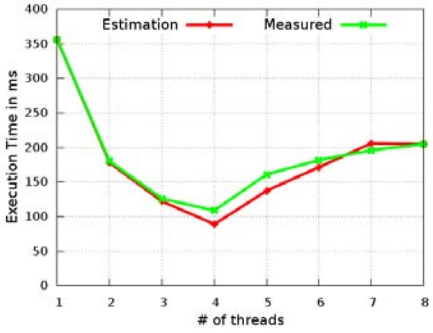
(b) MN3 – Add



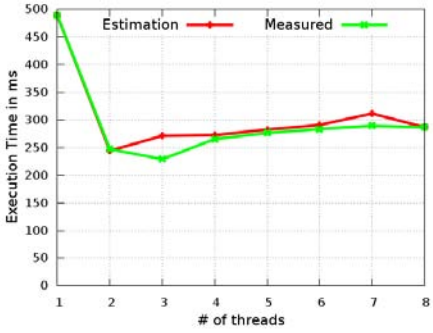
(c) SuperMUC Fat Node – Copy



(d) SuperMUC Fat Node – Add

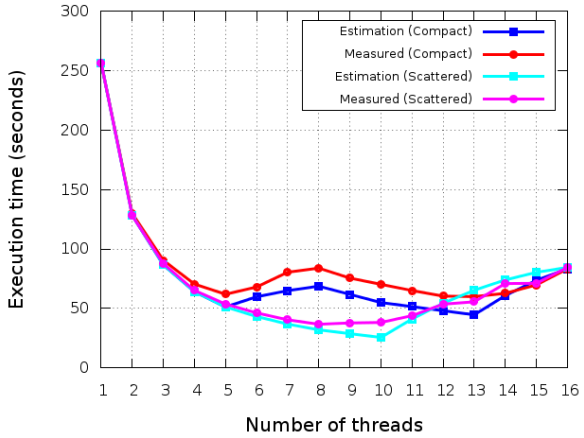


(e) SuperMUC Thin Node – Copy

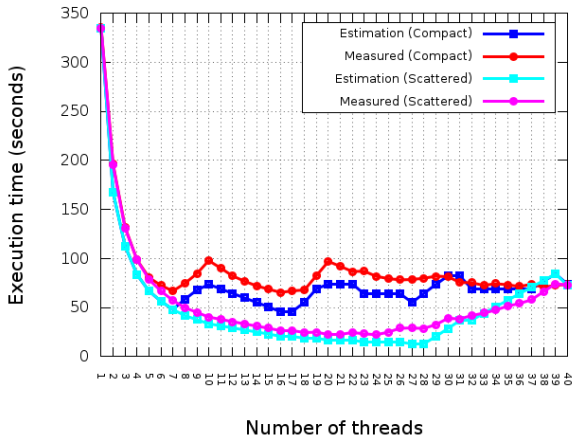


(f) SuperMUC Thin Node – Add

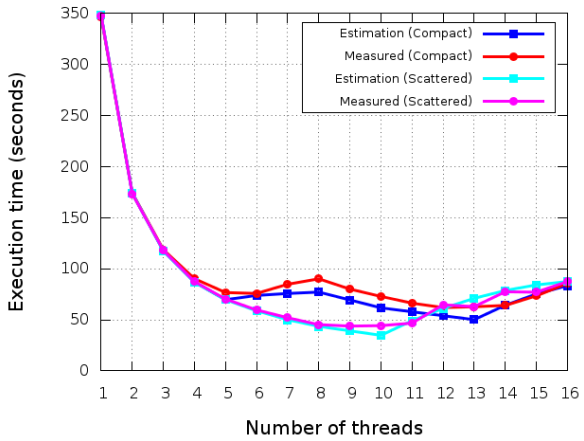
Figure 6.7: Comparison of model estimated execution times against real execution times on different architectures (MN3, SuperMUC Fat and Thin nodes) for parallel regions Copy and Add from the stream benchmark.



(a) MN3 system



(b) SuperMUC Fat node



(c) SuperMUC Thin node

Figure 6.8: Comparison of model estimated execution times against real execution times for parallel regions x_solve for the SP benchmark using distribution policies compact (AFF0) and scattered (AFF1).

Table 6.6: Speedup evaluation of the selected configuration compared with best configuration.

Benchmark	Parallel Region	MN3		Thin Node		Fat Node	
		`Selected`	`Best`	`Selected`	`Best`	`Selected`	`Best`
Stream	Add	*2th* 1.28x	*3th* 1.41x	*2th* 1.14x	*3th* 1.23x	*4th* 1.28x	*3th* 1.49x
Stream	Copy	*4th* 1.97x		*4th* 1.88x		*6th* 1.28x	*4th* 1.49x
SP.C	x_solve	*10th* 2.21x	*8th* 2.30x	*10th* 1.98x	*9th* 1.99x	*27th* 2.5x	*24th* 3.27x

6.6 Summary

The increasing number of cores in current multicore processors causes severe scalability problems in the cache hierarchy. Memory intensive applications can experience performance degradation on parallel region execution when increasing the number of threads for some architectures. The analysis of application characteristics can expose latent performance conflicts with the memory hierarchy.

In this chapter, we have defined a new model based on a pre-characterization of the application for small workloads with the aim of reducing the number of iterations required for characterizing the performance at runtime. This model is capable of estimating a convenient configuration of number of threads and thread distribution policy after 1 iteration of the application. We obtain a characterization of the memory footprint provided by a tracing the memory access pattern of the application, and we apply a performance model to provide a configuration of number of threads that minimizes performance degradation due to memory contention for any workload.

The SP application from the NAS Parallel Benchmarks, and a version of Stream Benchmark have been characterized using a tracing tool to obtains the memory trace on parallel regions. By analyzing the memory access pattern it is possible to obtain the most significant part of the memory which would benefit performance by preserving this memory on the LLC.

The methodology has been applied for different workloads of the applications and the estimations provided by the model have selected configuration minimizing the effect of performance degradation. The experimental results show a maximum 2.5x speedup on the class C compared with the default configuration using all the available threads.

7
Conclusions

"His soul swooned slowly as he heard the snow falling faintly through the universe and faintly falling, like the descent of their last end, upon all the living and the dead."

The dead (Dubliners) - **James Joyce**

This chapter presents the experiences gained and conclusions derived from this thesis. We also describe the viable open lines that can be considered in the future in order to provide further strategies and performance models in the area of dynamic tuning of OpenMP parallel applications.

7.1 Conclusions

The defining aspect of this thesis is the definition of a methodology in order to develop performance models and tuning strategies for OpenMP applications (Chapter 3). The methodology has been successfully applied on the context of a performance factor based on memory contention on multicore system architectures, and two performance models have been provided. A first model based on runtime characterization (Chapter 5) and a second model based on memory access pattern analysis from memory traces (Chapter 6).

We initially have had to identify a relevant performance factor based on performance degradation and thread imbalance. This has been done by characterizing and analyzing a representative set of applications of the HPC environments, such as the NAS parallel benchmarks suite. Through the performance analysis of this set of applications we have identified and evaluated the impact in performance on different hardware architectures. A tuning strategy and an initial performance model have been defined in Chapter 4

We have put our efforts on the most relevant performance factor detected which occurs in memory intensive applications such as the SP, MG, LU benchmarks. In these applications, memory contention is originated by the bottleneck generated between application's memory access patterns and the memory subsystem in the hardware architecture. The final effect is that performance is affected by memory contention, and furthermore, in some cases a performance degradation, like in the case of SP application.

In order to displace the memory contention bottleneck it would be desirable to provide a better architecture, however, this is not always possible, because sometimes the performance problem inherent to the context. Therefore, we have evaluated and observed the persistence of the performance memory contention problem on different available architectures, from workstations to highly scalable multicore systems.

Once the performance factor based on memory contention has proven to be relevant, we have considered the possibilities of alleviating the performance factor by setting a proper configuration of the environment. This is done by configuring the runtime execution using a proper number of threads and threads distribution along the shared memory system to avoid performance degradation.

Following this, it is possible to configure the application of number of threads and threads distribution to avoid performance degradation but, initially, this configuration it is not know. We have used an initial performance model based on a runtime exhaustive characterization in order to explore the possibilities of tuning the application at runtime. This approach can be performed in iterative applications by sacrificing the initial iterations to perform an evaluation

of every configuration. Once the characterization is finished, the best configuration is selected and the application tune for the rest of the execution. This strategy has been implemented using a function interposition library of the OpenMP interface. Even though the reported overheads were less than 5%, the characterization phase of the strategy is not scalable.

After evaluating the viability of using a tuning tool to modify the OpenMP runtime execution, a deeper analysis has been developed in order to define a performance model that provides a configuration of number of threads and thread distribution based on the analysis of last level cache misses (LLC˙misses).

The performance model based on runtime characterization (Chapter 5) reduces the characterization phase requirements by analyzing execution times and total cache misses in a single socket to provide the best configuration for high scalable systems such as the multicore multisocket nodes from clusters and supercomputer centres.

This model has been evaluated in the T7500 node at the UAB and on nodes FatNode and ThinNode from SuperMUC supercomputer at the LRZ centre. The model has been able to detect performance degradation by providing configurations improving the overall performance and with a maximum speedup of 2.74x for the 20 threads configuration using 5 threads per socket.

The defined performance model based on runtime characterization has some drawbacks. The model provides two estimation threshold and the real execution is supposed to be between these values. We have observed that the real execution tends to be close to one of the estimations. To determine the estimation it is possible to average this distance to determine a point in the middle. However, in some cases the distance between the boundary values is high making the estimation inaccurate.

To determine the closest threshold to provide a not supervised version of the model with the same degree of accuracy, we started analyzing the memory access pattern, which at the end turned into a different performance model.

The second model is based on a pre-characterization of the application using a tracing tool (pintool) developed with the PIN binary dynamic instrumentation tool (Chapter 6). This methodology requires a small set of traces to obtain the memory footprint based on the principle of identifying consecutive regions with a relevant ratio of reutilization.

We use the memory footprint to identify, for a concurrent execution, the amount of reaccessed data that benefits from being preserved in the LLC. The model uses the data occupancy size to determine where iteration time is going to degradate performance. The iteration time is reconstructed for the OpenMP loop construct by using a static scheduling

distribution of iteration per threads. This model also allows the exploration of different configurations, and through the exploration of different distribution affinities a configuration is provided. Finally, the selected configuration improves performance by minimizing the effect of memory over occupation and reducing the LLC misses.

This model has been evaluated on nodes from the Marenostrum supercomputer and Thin and Fat Nodes from SuperMUC supercomputer. The maximum speedup obtained is 2.5x using a 27 threads on an scattered configuration.

Finally, the results of both models have proven to detect memory contention and provide a configuration that improves performance on memory contention compared with a default execution using the maximum number of threads. Furthermore, the logic of the models can be easily integrated within the tuning tool based on library interposition with small changes.

7.2 Future Work

The work presented in this thesis will allow for further investigation into specific dynamic tuning techniques for OpenMP environments in multicore systems.

We have classified some relevant topics that need to considered in order to increase the scope of this research;

- **New Architectures**; The most directly related extension is to evaluate how to adapt or apply the models on system a architectures with high scalability, which can be designed with a different memory hierarchy subsystem such as Intel Xeon Phi. The Knights Landing processor provides 72 Silvermont-based cores in a 2D mesh interconnection network on chip.

  On the other hand, taking into consideration efficient computing, processors such as the big.LITTLE technology from ARM, which provides heterogeneous multicore processors with two sets of cores, a pair of more powerful Cortex-A15 out-of-order superscalar processors and a a couple of Cortex-A7 in-order processor. These systems would be benefited by applying scheduling strategies in order to select one or the other cores, depending on the requirements of the application, for example, in the case of memory contention.

- **Task parallelism**; In the development of this thesis we have considered and evaluated the possibility of using the performance model for improving the dynamic schedule of tasks. Our model's memory footprint has the same granularity as a task,

therefore it can be easily extended to memory task footprint. It is possible to use the memory footprint to label task depending on the memory requirements. In our case, the models limit the scalability of the application in order to relieve the concurrent footprint at LLC. However, the concept of memory footprint can be extended by scheduling compute intensive tasks on the unused cores.

- **Programming model**; Following the previous idea, the concept of memory footprint can also be used in the scope of distributed memory models such as MPI (shared memory MPI_threads). The model can also be extended to the upper memory subsystem with the aim of limiting the memory swapping on disk.

- **Simulation**; Simulation can be considered as the space between emulation and performance modeling. A simulator could let the user to decide the accuracy and precision of its simulation, at the cost of time. Therefore, some simulators assume some simplifications based on using performance models to estimate some behaviours. This sacrifices accuracy but also decrease the simulation time. We believe our model based on memory footprint can provide some help in order to perform simulations. As an example, our memory tracing tool is based on the TaskSim simulation infrastructure [85] for simulating applications based on the OmpSs programming model, and could be easily integrated within.

- **Hardware support**; During the development of this thesis we have seen that hardware counters are not always helpful in order to understand the application's behaviour. We believe that hardware performance monitoring tools can provide far more complex hints and orientations. We have discussed the possibility of developing such elements, for example, in our case, to provide a hardware support unit to provide the memory footprint at runtime with a small overhead.

7.3 List of publications

The work and motivation for this thesis have been published in the following papers:

1. **C.Allande, J.Jorba, E.César, and A.Sikora. "Metodología para la sintonización de aplicaciones OpenMP en entornos multicore" in XXII Jornadas de Paralellismo, pp. 649-654. 2011.**

In this paper, we define a methodology to identify performance factors for OpenMP applications. We present performance factors based on scheduling strategies for iterative parallel loop and tasks parallel applications. Data parallelism is analyzed using an embarrassingly parallel application based on image segmentation. In addition, performance factors for task parallelism are studied using different scheduling strategies on the recursive generation of high umbalance task tree.

2. **C.Allande, J.Jorba, A.Sikora, and E.César. "A Methodology for generating Dynamic Tuning strategies in Multicore Systems" in Proceeding of the International Conference on Parallel and Distributed Processing Techniques and Applications, Volume II pp 780-786, 2012.**

   This paper presents a methodology to systematically develop performance optimization strategies for specific application patterns taking into consideration architecture characteristics. This describes the methodology developed and shows how it can be used to expose performance factors that can be dynamically tuned on an OpenMP application.

3. **C.Allande, J.Jorba, A.Sikora, and E.César. "A Performance Model for OpenMP Memory Bound Applications in Multisocket Systems", in Proceeding of the International Conference on Computational Science, Volume 29, pp.2208-2218, 2014.**

   In this paper, we present a performance model to select the number of threads and affinity distribution for parallel regions on OpenMP applications executed in multisocket multicore processors. The model is based on a characterization of the performance of parallel regions to estimate cache misses and execution time. Estimated execution time is used to select a set of configurations in order to minimize the impact of memory contention, achieving significant improvements compared with a default configuration using all threads.

4. **C.Allande, J.Jorba, A.Sikora, E.César. "Performance model based on memory footprint for OpenMP memory bound applications", to appear in Proceeding of the International Conference on Parallel Computing. 2015.**

   This paper presents a performance model to estimate the execution time for a number of thread and affinity distribution for an OpenMP application parallel region based on runtime hardware counters information and the estimation of performance

degradation due to memory contention generated at last level cache (LLC). The model considers features obtained from the memory access pattern and the memory footprint. The proposed methodology identifies the thread configurations which maximize the application performance by preventing memory contention on main memory.

7.4 Acknowledgements

This work has been partially supported by the MICINN-Spain under contracts TIN2007-64974, TIN2011-28689 and TIN2012-34557, the European Research Council under the European Union's 7th FP, the ERC Grant Agreement number 321253, and the predoctoral research grants (PIF 2009-2010) from the CAOS department at the UAB.

In addition, the following grants were received during the course of this work:

- ICTS 2012; access to the resources of the High Performance Computing resouces at the CEntro de Supercomputación de GAlicia (CESGA) for the project "Sintonización dinámica de aplicaciones en entornos de memoria compartida".
- BE-DGR 2012, "Beca para estancias predoctorales de corta duración en el extranjero" provided by the Agencia de Gestió i d'Ajuts Universitaris i de Recerca (AGAUR). Part of this grant involved a research stay at the Munich Network Management Team (MNM-Team) at the Ludwig-Maximillians-Universität (LMU).

The authors thankfully acknowledge the resources and technical assistance provided by Munich Network Management Team (MNM-Team), the Leibniz Supercomputing Centre (LRZ), the Barcelona Supercomputing Center (BSC), DPCS-UOC (Distributed, Parallel and Collaborative Systems Research Group) from Universitat Oberta de Catalunya (UOC), and the CAOS department at the UAB.

Bibliography

[1] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "Tile64 - processor: A 64-core soc with mesh interconnect," in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, Feb 2008, pp. 88–598.

[2] A. Duran and M. Klemm, "The intel many integrated core architecture," in *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, July 2012, pp. 365–366.

[3] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46 –55, jan-mar 1998.

[4] C. Pheatt, "Intel® threading building blocks," *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 298–298, Apr. 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1352079.1352134

[5] "The openacc application programming interface, version 2.0." http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf, accessed: 2015-10-1.

[6] R. Blumofe and et al., "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, pp. 207–216, August 1995.

[7] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer, "Modeling performance variation due to cache sharing," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, Feb 2013, pp. 155–166.

[8] C.-K. Luk and et al., "Pin: building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, pp. 190–200, June 2005.

[9] G. Lee and et al., "Dynamic binary instrumentation and data aggregation on large scale systems," *International Journal of Parallel Programming*, vol. 35, pp. 207–232, 2007.

[10] "Top500 supercomputer list," http://http://www.top500.org/, accessed: 2015-10-1.

[11] J. Tendler, J. Dodson, J. Fields, H. Le, and B. Sinharoy, "Power4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, Jan 2002.

[12] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[13] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos mercurium: a research compiler for openmp," in *Proceedings of the European Workshop on OpenMP*, vol. 8, 2004.

[14] "Bsc performance tools, paraver internals and details." http://www.bsc.es/ssl/apps/performanceTools/files/docs/W2_Paraver_details.pdf, accessed: 2015-09-22.

[15] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: Transputer and occam Developments*, vol. 44. mar, 1995, pp. 17–31.

[16] "Intel xeon phi coprocessor 5110p. 60 core." http://ark.intel.com/es-es/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core, accessed: 2015-09-14.

[17] F. Baskett, T. Jermoluk, and D. Solomon, "The 4d-mp graphics superworkstation: computing+graphics=40 mips+mflops and 100000 lighted polygons per second," in *Compcon Spring '88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers*, Feb 1988, pp. 468–471.

[18] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," *SIGARCH Comput. Archit. News*, vol. 12, no. 3, pp. 348–354, Jan. 1984. [Online]. Available: http://doi.acm.org/10.1145/773453.808204

[19] G. Delzanno, "Automatic verification of parameterized cache coherence protocols," in *Computer Aided Verification*. Springer, 2000, pp. 53–68.

[20] J. R. Goodman and H. H. J. Hum, "Mesif: A two-hop cache coherency protocol for point-to-point interconnects," *University of Auckland, Tech. Rep*, 2009.

[21] C. P. Thacker and L. C. Stewart, "Firefly: A multiprocessor workstation," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 4, pp. 164–172, Oct. 1987. [Online]. Available: http://doi.acm.org/10.1145/36204.36199

[22] E. M. McCreight, "The dragon computer system," in *Microarchitecture of VLSI Computers*. Springer, 1985, pp. 83–101.

[23] W. Starke, J. Stuecheli, D. Daly, J. Dodson, F. Auernhammer, P. Sagmeister, G. Guthrie, C. Marino, M. Siegel, and B. Blaner, "The cache and memory subsystems of the ibm power8 processor," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 3:1–3:13, Jan 2015.

[24] A. Intel, "Introduction to the intel quickpath interconnect," *White Paper*, 2009.

[25] H. T. Consortium *et al.*, "Hypertransport 1," *O Link Specification*, 2003.

[26] "big.little technology: The future of mobile," https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf, accessed: 2015-10-1.

[27] "Mont-blanc, european approach towards energy efficient high performance," https://www.montblanc-project.eu/, accessed: 2015-10-1.

[28] "The openmp specification, version 3.0." http://www.openmp.org/mp-documents/spec30.pdf, accessed: 2015-10-1.

[29] "About the openmp arb and openmp.org." http://openmp.org/wp/about-openmp/, accessed: 2015-09-22.

[30] D. P. Siewiorek and P. J. Koopman, *The architecture of supercomputers: Titan, a case study*. Academic Press, 2014.

[31] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[32] J. McCalpin, "Stream benchmark," *Link: www. cs. virginia. edu/stream/ref. html# what*, 1995.

[33] "Lawrence livermore national laboratory. stride sequoia benchmarks source code." https://asc.llnl.gov/sequoia/benchmarks/, accessed: 2015-09-22.

[34] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[35] J. M. Bull and D. O'Neill, "A microbenchmark suite for openmp 2.0," *SIGARCH Comput. Archit. News*, vol. 29, pp. 41–48, December 2001.

[36] A. Marowka, "Empirical analysis of parallelism overheads on cmps," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Springer Berlin Heidelberg, 2010, vol. 6067, pp. 596–605. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14390-8_62

[37] C. Liao, Z. Liu, L. Huang, and B. Chapman, "Evaluating openmp on chip multithreading platforms," in *OpenMP Shared Memory Parallel Programming*, ser. Lecture Notes in Computer Science, M. Mueller, B. Chapman, B. de Supinski, A. Malony, and M. Voss, Eds. Springer Berlin Heidelberg, 2008, vol. 4315, pp. 178–190. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68555-5_15

[38] A. Duran and et al., "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *Parallel Processing, 2009. ICPP '09. International Conference on*, sept. 2009, pp. 124 –131.

[39] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[40] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," *CoRR*, vol. abs/1004.4431, 2010.

[41] A. Kleen, "A numa api for linux," *Novel Inc*, 2005.

[42] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*. New York, NY, USA: ACM, 1982, pp. 120–126.

[43] S. Godard, "Sysstat: System performance tools for the linux os, 2004."

[44] K. Fürlinger and M. Gerndt, "ompp: A profiling tool for openmp," in *OpenMP Shared Memory Parallel Programming*, ser. Lecture Notes in Computer Science, M. Mueller, B. Chapman, B. Supinski, A. Malony, and M. Voss, Eds. Springer Berlin Heidelberg, 2008, vol. 4315, pp. 15–23. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-68555-5_2

[45] J. Vetter and C. Chambreau, "mpip: Lightweight, scalable mpi profiling," *URL http://mpip. sourceforge. net*, 2005.

[46] J. Mellor-Crummey, R. J. Fowler, G. Marin, and N. Tallent, "Hpcview: A tool for top-down analysis of node performance," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 81–104, 2002.

[47] R. Kufrin, "Perfsuite: An accessible, open source performance analysis environment for linux," in *6th International Conference on Linux Clusters: The HPC Revolution*, vol. 151. Citeseer, 2005, p. 05.

[48] J. Borrill, J. Carter, L. Oliker, D. Skinner, and R. Biswas, "Integrated performance monitoring of a cosmology application on leading hec platforms," in *Parallel Processing, 2005. ICPP 2005. International Conference on*. IEEE, 2005, pp. 119–128.

[49] W. Gropp and K. Buschelman, "Fpmpi-2 fast profiling library for mpi," *www-unix. mcs. anl. gov/fpmpi*, vol. 8, 2006.

[50] "Intel vtune amplifier." https://software.intel.com/en-us/intel-vtune-amplifier-xe, accessed: 2015-09-22.

[51] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel, "Developing scalable applications with vampir, vampirserver and vampirtrace." in *PARCO*, vol. 15. Citeseer, 2007, pp. 637–644.

[52] "Bsc performance tools, extrae." http://www.bsc.es/computer-sciences/extrae, accessed: 2015-09-22.

[53] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "Vampir: Visualization and analysis of mpi resources," *Supercomputer*, vol. 12, pp. 69–80, 1996.

[54] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, "Open— speedshop: An open source infrastructure for parallel performance analysis," *Scientific Programming*, vol. 16, no. 2-3, pp. 105–121, 2008.

[55] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[56] S. S. Shende and A. D. Malony, "The tau parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, pp. 287–331, 2006.

[57] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[58] E. Cesar, A. Moreno, J. Sorribes, and E. Luque, "Modeling master/worker applications for automatic performance tuning," *Parallel Computing*, vol. 32, no. 7?8, pp. 568 – 589, 2006, algorithmic Skeletons. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819106000263

[59] A. Guevara, E. Cesar, J. Sorribes, T. Margalef, E. Luque, and A. Moreno, "A performance tuning strategy for complex parallel application," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*. IEEE, 2010, pp. 103–110.

[60] A. Morajko, A. Martínez, E. César, T. Margalef, and J. Sorribes, "Mate: toward scalable automated and dynamic performance tuning environment," in *Applied Parallel and Scientific Computing*. Springer, 2012, pp. 430–440.

[61] A. Morajko, O. Morajko, T. Margalef, and E. Luque, "Mate: Dynamic performance tuning environment," in *Euro-Par 2004 Parallel Processing*. Springer, 2004, pp. 98–107.

[62] C. Ţăpuş, I.-H. Chung, and J. K. Hollingsworth, "Active harmony: Towards automated performance tuning," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, ser. SC '02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=762761.762771

[63] R. L. Ribler, H. Simitci, and D. A. Reed, "The autopilot performance-directed adaptive control system," *Future Generation Computer Systems*, vol. 18, no. 1, pp. 175 – 187, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/B6V06-43V9S99-K/2/ef6ab9580b90ed17bc2c5d29e62b882b

[64] B. Wicaksono and et al., "A dynamic optimization framework for openmp," in *OpenMP in the Petascale Era*, ser. LLNCS. Springer Berlin / Heidelberg, 2011, vol. 6665, pp. 54–68.

[65] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The Nas Parallel Benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991. [Online]. Available: http://online.sagepub.comhttp://hpc.sagepub.com/content/5/3/63.abstract

[66] S. Olivier and et al., "Uts: An unbalanced tree search benchmark," in *Languages and Compilers for Parallel Computing*, ser. LLNCS. Springer Berlin / Heidelberg, 2007, vol. 4382, pp. 235–250.

[67] A. Duran, J. Corbalan, and E. Ayguade, "An adaptive cut-off for task parallelism," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, nov. 2008, pp. 1 –11.

[68] S. Olivier and J. Prins, "Evaluating openmp 3.0 run time systems on unbalanced task graphs," in *Evolving OpenMP in an Age of Extreme Parallelism*, ser. LLNCS. Springer Berlin / Heidelberg, 2009, vol. 5568, pp. 63–78.

[69] S. Williams and et al., "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.

[70] K. Fuerlinger and M. Gerndt, "Analyzing overheads and scalability characteristics of openmp applications," in *High Performance Computing for Computational Science-VECPAR 2006*. Springer, 2007, pp. 39–51.

[71] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier, "Dynamic task and data placement over numa architectures: An openmp runtime perspective," in *Evolving OpenMP in an Age of Extreme Parallelism*, ser. Lecture Notes in Computer Science, M. Müller, B. de Supinski, and B. Chapman, Eds.

Springer Berlin Heidelberg, 2009, vol. 5568, pp. 79–92. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02303-3_7

[72] J. Corbalan, A. Duran, and J. Labarta, "Dynamic load balancing of mpi+openmp applications," in *Parallel Processing, 2004. ICPP 2004. International Conference on*, Aug 2004, pp. 195–202 vol.1.

[73] J. Li, J. Shu, Y. Chen, D. Wang, and W. Zheng, "Analysis of factors affecting execution performance of openmp programs," *Tsinghua Science & Technology*, vol. 10, no. 3, pp. 304–308, 2005.

[74] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of openmp task scheduling strategies," in *OpenMP in a new era of parallelism*. Springer, 2008, pp. 100–110.

[75] B. Tudor and Y.-M. Teo, "A practical approach for performance analysis of shared-memory programs," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 652–663.

[76] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proceedings of the 11th Int. Symp. on HPCA*, ser. HPCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 340–351.

[77] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 129–142.

[78] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei, "A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads," in *Proceedings of the ICHPC, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 83:1–83:11.

[79] M. Ghosh, R. Nathuji, M. Lee, K. Schwan, and H. Lee, "Symbiotic scheduling for shared caches in multi-core systems using memory footprint signature," in *Parallel Processing (ICPP), 2011 International Conference on*, Sept 2011, pp. 11–20.

[80] S. Biswas, B. De Supinski, M. Schulz, D. Franklin, T. Sherwood, and F. Chong, "Exploiting data similarity to reduce memory footprints," in *IPDPS, 2011 IEEE International*, May 2011, pp. 152–163.

[81] S. Jana and V. Shmatikov, "Memento: Learning secrets from process footprints," in *Security and Privacy (SP), 2012 IEEE Symposium on*, May 2012, pp. 143–157.

[82] C. Ding, X. Xiang, B. Bao, H. Luo, Y.-W. Luo, and X.-L. Wang, "Performance metrics and models for shared cache," *Journal of Computer Science and Technology*, vol. 29, no. 4, pp. 692–712, 2014.

[83] B. Brett, P. Kumar, M. Kim, and H. Kim, "Chip: A profiler to measure the effect of cache contention on scalability," in *IPDPSW, 2013 IEEE 27th International*, May 2013, pp. 1565–1574.

[84] J. R. Tramm and A. R. Siegel, "Memory bottlenecks and memory contention in multi-core monte carlo transport codes," *Annals of Nuclear Energy*, 2014.

[85] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero, "Trace-driven simulation of multithreaded applications," in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 87–96.