Contributions au contrôle de l'affinité mémoire sur architectures multicoeurs et hiérarchiques

Christiane Pousa Ribeiro

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Christiane Pousa Ribeiro

Thèse dirigée par **Jean-François Méhaut**
et codirigée par **Alexandre Carissimi**

préparée au sein **Laboratoire d'Informatique de Grenoble**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Contributions on Memory Affinity Management for Hierarchical Shared Memory Multi-core Platforms

Thèse soutenue publiquement le **29-06-2011**,
devant le jury composé de :

Brésil, Philippe O. A. Navaux
Professor, University Federal of Rio Grande do Sul - Institute of Informatics, Présidente
France, Bernard Tourancheau
Professeur, Université de Lyon 1, Rapporteur
France, Raymond Namyst
Professeur, Université de Bordeaux I, Rapporteur
France, Dimitri Komatitsch
Professeur, Université de Toulouse - IUF, Examinateur
Etats-Unis, Laxmikant V. Kale
Professeur, University of Illinois at Urbana-Champaign, Examinateur
Brésil, Alexandre da Silva Carissimi
Professeur, University Federal of Rio Grande do Sul - Institute of Informatics, Co-Directeur de thèse
France, Jean-François Méhaut
Professeur, University of Grenoble - INRIA, Directeur de thèse

"What we anticipate seldom occurs, but what we least expect generally happens."
(Benjamin Disraeli)

Acknowledgments

First, I would like to thank my advisor Jean-François Méhaut for having accepted me as his PhD. student. For all the discussions, ideas and for the motivation throughout this PhD. I appreciate his patience at the beginning of my PhD. due to my difficulties with the language. Also, the incredible opportunities I have had during this thesis, the projects, meetings, conferences and collaborations. I would also like to thank you for always encouraging cooperation between Brazil and France and for giving me the opportunity to work on projects between the two countries.

I also would like to thank my co-advisor Alexandre Carissimi, for his scientific and technical support, all the corrections and suggestions in all stages of my thesis. Many e-mails were exchanged in those three years and a half, with ideas, reflections and questions that helped me in doing my work. Thanks also for all the tips about France. Thank you Carissimi, your support was fundamental in the development of this work.

I would like to thank all members of the defense jury: Raymond Namyst, Bernard Tourancheau, Dimitri Komatitsch, Laxmikant Kale and especially Philippe O. A. Navaux for being the chair on my defense and having followed the whole process of this work, since the selection for the scholarship in Brazil until the PhD. defense.

This thesis would not be possible without funding from CAPES-Brazil and the French and Brazilian projects that made possible my travels in all meetings and conferences which I attended.

Thanks to the professors of LIG (Laboratoire d'Informatique de Grenoble), particularly the ones from Mescal and Moais teams. For all discussions and conversations over the years of my thesis. Thanks for all the teachings. Special thanks to Jean-Marc Vincent for the statistical insights and to Vania Marangozova-Martin for our conversations and laughter that made my life so much easier and pleasant in France.

I would also like to thank all my friends from LIG, BRGM, UFRGS and PUC-Minas, for all support ;). There were many talks and chats at the 'Kafet' of our laboratory that helped me to achieve this thesis. Special thanks to Fabrice Dupros for the initial motivation of my work with 'his' application issues, to the 'Lucas and Fabiane', 'Daniel and Kelly', 'Luis Fabrício and Raquel', 'Dulcinéia and Raphael' for the friendship and support. ... you are special!; to Rodrigue Chakode for his patience to help me with French; to Márcio and Laércio for all work that we did together and to all my Brazilians friends in Grenoble who have made these years in France easier.

A special thanks to all of my family in Brazil, for the love and for cheering me up! Thank you for supporting my decisions and understanding that this was my dream. Thank you for the emails and skype sessions, for all the help and encouraging words you gave me. An extra special thanks to my Mom, Rosilene, and my brother, Paulo Júneo. This achievement is also for you!

Finally, a big thanks to Rodrigo Ribeiro or 'Fofinho', for all the love, comfort and support during these years of studies. These were harsh years because of the distance and solitude, but his words gave me the courage to move on and his love was my main motivation to complete this dream. Thank you for leaving behind a lifetime in Brazil and come follow me in this dream, thanks for getting me up every time I get discouraged ... Thank you Fofinho.

Abstract:
Multi-core platforms with non-uniform memory access (NUMA) design are now a common resource in High Performance Computing. In such platforms, the shared memory is organized in an hierarchical memory subsystem in which the shared memory is physically distributed into several memory banks. Additionally, these platforms feature several levels of cache memories. Because of such hierarchy, memory access latencies may vary depending on the distance between cores and memories. Furthermore, since the number of cores is considerably high in these machines, concurrent accesses to the same memory banks are performed, degrading bandwidth usage. Therefore, a key element in improving the application performance on these machines is dealing with memory affinity.
Memory affinity is a relationship between threads and data of application that describes how threads access data. In order to keep memory affinity a compromise between data and thread placement is then necessary. In this context, the main objective of this thesis is to attain scalable performances on multi-core NUMA machines by reducing latencies and increasing memory bandwidth. The first goal of this thesis is to investigate which characteristics of the NUMA platform and the application have an important impact on the memory affinity control and propose mechanisms to deal with them on multi-core machines with NUMA design. We focus on High Performance Scientific Numerical workloads with regular and irregular memory access characteristics. The study of memory affinity aims at the proposal of an environment to manage memory affinity on Multi-core Platforms with NUMA design. This environment provides fine grained mechanisms to manage data placement for an application using the application compile time information, runtime information and architecture characteristics.
The second goal is to provide solutions that show performance portability. We mean by performance portability, solutions that are capable of providing similar performances on different NUMA platforms. To do so, we propose mechanisms that are independent of machine architecture and compiler. The portability of the proposed environment is evaluated through the performance analysis of several benchmarks and applications over different platforms.
Finally, the third goal of this thesis is to implement memory affinity mechanisms that can be easily adapted and used in different parallel systems. Our approach takes into account the different data structures used in High Performance Scientific Numerical workloads, in order to provide solutions that can be used in different contexts. All the ideas developed in this research work are implemented within a Framework named Minas (Memory affInity maNAgement Software). We evaluate the applicability of such mechanisms in three parallel programming systems, OpenMP, Charm++ and OpenSkel. Additionally, we evaluated Minas performance using several benchmarks and two real world applications from geophysics.

Résumé:

Les plates-formes multi-cœurs avec un accès mémoire non uniforme (NUMA) sont devenu des ressources usuelles de calcul haute performance. Dans ces plates-formes, la mémoire partagée est constituée de plusieurs bancs de mémoires physiques organisés hiérarchiquement. Cette hiérarchie est également constituée de plusieurs niveaux de mémoires caches et peut être assez complexe. En raison de cette complexité, les coûts d'accès mémoire peuvent varier en fonction de la distance entre le processeur et le banc mémoire accédé. Aussi, le nombre de cœurs est très élevé dans telles machines entraînant des accès mémoire concurrents. Ces accès concurrents conduisent à des ponts chauds sur des bancs mémoire, générant des problèmes d'équilibrage de charge, de contention mémoire et d'accès distants. Par conséquent, le principal défi sur les plates-formes NUMA est de réduire la latence des accès mémoire et de maximiser la bande passante.

Dans ce contexte, l'objectif principal de cette thèse est d'assurer une portabilité de performances sur des machines NUMA multi-cœurs en contrôlant l'affinité mémoire. Le premier aspect consiste à étudier les caractéristiques des plates-formes NUMA que sont à considérer pour contrôler efficacement les affinités mémoire, et de proposer des mécanismes pour tirer partie de telles affinités. Nous basons notre étude sur des benchmarks et des applications de calcul scientifique ayant des accès mémoire réguliers et irréguliers. L'étude de l'affinité mémoire nous a conduit à proposer un environnement pour gérer le placement des données pour les différents processus des applications. Cet environnement s'appuie sur des informations de compilation et sur l'architecture matérielle pour fournir des mécanismes à grains fins pour contrôler le placement.

Ensuite, nous cherchons à fournir des solutions de portabilité des performances. Nous entendons par portabilité des performances la capacité de l'environnement à apporter des améliorations similaires sur des plates-formes NUMA différentes. Pour ce faire, nous proposons des mécanismes qui sont indépendants de l'architecture machine et du compilateur. La portabilité de l'environnement est évaluée sur différentes plates-formes à partir de plusieurs benchmarks et des applications numériques réelles.

Enfin, nous concevons des mécanismes d'affinité mémoire qui peuvent être facilement adaptés et utilisés dans différents systèmes parallèles. Notre approche prend en compte les différentes structures de données utilisées dans les différentes applications afin de proposer des solutions qui peuvent être utilisées dans différents contextes. Toutes les propositions développées dans ce travail de recherche sont mises en œuvre dans une framework nommée Minas (Memory Affinity Management Software). Nous avons évalué l'adaptabilité de ces mécanismes suivant trois modèles de programmation parallèle à savoir OpenMP, Charm++ et mémoire transactionnelle. En outre, nous avons évalué ses performances en utilisant plusieurs benchmarks et deux applications réelles de géophysique.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, a large number of numerical scientific applications in different areas of knowledge (geophysics, meteorology, etc) demand low response times and high memory capacity. Due to these needs, those applications require powerful computational resources to obtain high performances. Particularly, high performance computing (HPC) platforms such as multi-core machines can provide the necessary resources for numerical scientific applications. A multi-core machine is a multi-processed system in which the processing units (cores) are encapsulated into processors that share different levels of cache memories and the global main memory.

On multi-core machines the number of cores per sockets keeps increasing, leading to poor memory scalability and performance, the memory wall problem. The efforts to overcome this problem generally rely on the use of Non-Uniform Memory Access design within the architecture. Due to this, multi-core machines with NUMA design are becoming very common computing resources for HPC. On such machines, the shared memory is physically distributed over the machine and interconnected by a special network. Although this design preserves the abstraction of a single shared memory to the cores, it comes at the cost of load-balancing issues, memory contention and remote accesses.

In order to guarantee high performances on these architectures, an efficient data and thread placement strategy becomes crucial. Such placement can be achieved by enhancing memory affinity for parallel applications. Memory affinity is the relationship between threads and data of an application that describes how threads access data to accomplish a job. Therefore, in order to respect memory affinity a compromise between threads and data placement must be made to reduce latency and increase bandwidth for memory accesses.

A number of mainstream manufactures, operating systems and parallel languages concerns in providing efficient performances for multi-core NUMA machines. However, they lack in providing memory affinity solutions for the applications running on such machines. Consequently, developers are obliged to implement data and thread placement solutions to enhance memory affinity for applications and consequently, performance on NUMA machines. Besides, these placement solutions are implemented by developers using low level libraries and must suit their application and the targeted machine. Therefore, different versions of the same application source code are produced by the programmer, in order to obtain good performances on different platforms.

1.1 Objectives and Thesis Contributions

Due to the problems described in the previous section, it is important to investigate and understand the impact of memory affinity on parallel applications over current NUMA platforms. It is also important to study the affinity support that already exists for NUMA multi-core machines. We will notice on the next chapters that most of the available support for obtaining memory affinity is not standard. They demand complex application source code transformations from the developers and special operating system support that depends on platform characteristics. Furthermore, such support does not provide different levels to control memory affinity such as management of the application variables or heap.

Since parallel applications have different characteristics and needs, in order to enhance memory affinity for these applications an efficient data and thread placement strategy must be used. Therefore, it becomes necessary the proposal of a solution that matches the machine and the application characteristics in order to employ an efficient and portable memory affinity mechanism. We then propose an environment to manage memory affinity on multi-core platforms with NUMA design. This environment provides the necessary support to express data affinity in different levels (e.g. per variable, per heap or per application). During the execution time the environment maps the application information into the hardware characteristics. In this way, developers can have a NUMA-aware application that takes in consideration its memory access patterns and the platform characteristics.

Considering the objectives described in the above paragraphs, the main contributions of this thesis are: (I) a model that defines and characterizes a NUMA platform; (II) the design and the implementation of a framework to manage memory affinity for parallel applications in an automatic and explicit way, named Minas; (III) the integration of the framework mechanisms in different parallel applications (based on OpenMP, Charm++, AMPI and Software Transactional Memory) and parallel programming systems (Charm++ and OpenSkel).

1.2 Scientific Context

This thesis is developed under the project ANR NUMASIS [1] and Joint-Laboratory Illinois-INRIA-NCSA [2]. NUMASIS is a cooperation between BULL, BRGM, CEA and INRIA whereas the Joint-Laboratory is a cooperation between University of Illinois at Urbana-Champaign and INRIA. These organisms and enterprises have worked together in the project to design efficient solutions for High Performance Computing.

The scientific context of the NUMASIS project is the high performance computing in the geophysics domain. In this context, the problems arise from the optimization of parallel applications that implements seismic waves propagation

1. Adaptation et optimisation des performances applicatives sur architectures NUMA: Etude et mise en oeuvre sur des applications en SISmologie - URL: http://numasis.gforge.inria.fr

2. Joint Laboratory for Petascale Computing - URL:http://jointlab.ncsa.illinois.edu/

simulation on NUMA platforms. Results obtained from this project have helped BULL, BRGM and CEA in the design of new solutions for multi-core machines with NUMA characteristics.

The context of the Joint-Laboratory scientific is the software challenges for petascale computing. Problems that are considered by this Joint-Laboratory are related to modeling and optimizing numerical libraries, to fault-tolerance issues and to novel programming models. Results obtained from this initiative have helped both countries to improve software for high performance computers.

The research domain of these projects is in the context of the Laboratoire d'Informatique de Grenoble in which the Mescal research team is. Our role in the projects is the study the impact of memory affinity on parallel applications. In light of these studies, to propose new software solutions for the memory affinity management in parallel applications.

1.3 Thesis Organization

The thesis document is organized in three parts. The first part, subdivided in two chapters, introduces the scientific context of this work. In the first chapter, we describe Non-Uniform Memory Access (NUMA) multi-core platforms and its main architectural characteristics. That chapter also presents the relationship between such characteristics and the performance of the multi-core NUMA platform. In the second chapter the software components of a NUMA platform are introduced, focusing on the impact of such platform on the parallel systems, environments and applications.

The second part of the document introduces and describes the main contributions of the thesis and is subdivided in three chapters. The first of them describes the main ideas of the contributions, presenting the proposed strategies to improve memory affinity on hierarchical multi-core machines. It also illustrates the retrieval and modeling of the platform and applications characteristics. The second chapter of this part shows the design and implementation details of the proposed framework to control memory affinity. Finally, the third chapter shows how to employ our solutions in different high level parallel programming systems.

The third part presents a performance evaluation of the framework and its components, subdivided in four chapters. The first chapter describes the experimental methodology presenting the platforms and metrics used in our evaluation. In its second chapter, we show the performance evaluation of the framework components on OpenMP benchmarks and two real geophysics applications. The third chapter depicts the performance evaluation of the framework in benchmarks developed with some high level parallel programming systems such as Charm++, AMPI and OpenSkel. Finally, the last chapter points out our conclusions and the perspectives.

Part I

State of Art: Memory Affinity on Hierarchical Multi-core Platforms

Chapter 2

Hierarchical Shared Memory Multi-core Architectures

The current trend in high performance computing to obtain scalable performances is to increase the number of cores available per processor on shared memory machines. The multi-core chip design and the efforts to overcome the hardware limitations of classical Symmetric Multiprocessors (SMP) parallel systems have led to the emergence of the hierarchical architectures. These architectures feature a complex topology and a hierarchical memory subsystem. In this chapter, we introduce the state of the art in hierarchical shared memory multi-core architectures. We first provide the definition of what we consider a hierarchical shared memory architecture, discussing its main characteristics and how the hierarchy is designed inside the machine. After that, we discuss the main evolutions of shared memory multiprocessors, describing the multi-core architectures and the non-uniform memory access (NUMA) characteristics of current multiprocessors. We finish the chapter with a description of the hardware characteristics of hierarchical architecture memory subsystems.

2.1 What is a Hierarchical Shared Memory Architecture?

In this thesis, we consider as a hierarchical shared memory architecture any multiprocessor platform that features: (i) processing units that share a global memory and (ii) processing units and memory components organized in some hierarchical topology. In this context, examples of hierarchical shared memory architectures are: UMA (Uniform Memory Access) machines, NUMA (Non-Uniform Memory Access) machines, COMA (Cache Only Memory access machines) and NUCA (Non-Uniform Cache Access) [Patterson 2009].

In the UMA platforms all processing units have similar access costs to the global shared memory. This is due to the fact that the global shared memory is connected to a single bus that is used by processing units to access memory. Additionally, in this architecture the processing elements share the peripheral devices, which are also connected to the single bus. The main problem of this design is that the bus becomes a bottleneck, since all processing units must use it to access the global memory and the peripheral devices. Therefore, such bus restricts the scalability of the UMA architecture. We consider this architecture as a hierarchical shared

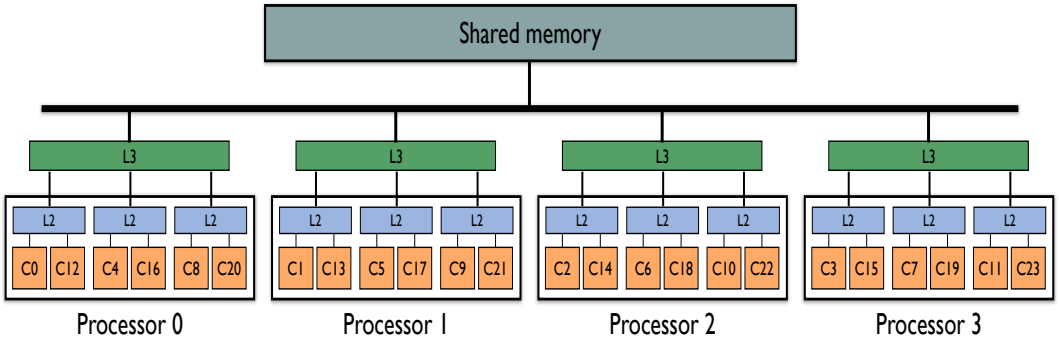


Figure 2.1: The UMA Architecture.

memory multiprocessor because of the current core topology designed within these machines.

Current UMA platforms feature a complex topology, with multiple processors, multi-core chips and cache memories. Figure 2.1 shows a multi-core platform with UMA design. In the figure, we can observe that the machine has four processors and each one has six cores. Considering the memory subsystem, the machine has two levels of shared cache memory. Each pair of cores share a L2 cache memory and each processor has a shared L3 cache memory. Main memory is shared between all cores of the machine and accessed by all cores through a single bus. Even tough these machines have an uniform access to the shared memory, it is important to take into account the topology when mapping the application process/threads. Due to the hierarchical organization of cores, processors and cache memories, communication time between processing units may change, depending on the distance between them [Mei 2010, Cruz 2010]. For instance, in the machine presented in the Figure 2.1, cache hierarchy can be explored to reduce the communication time between a group of threads. Such group can be placed within the same processor, avoiding the inter-processor communication.

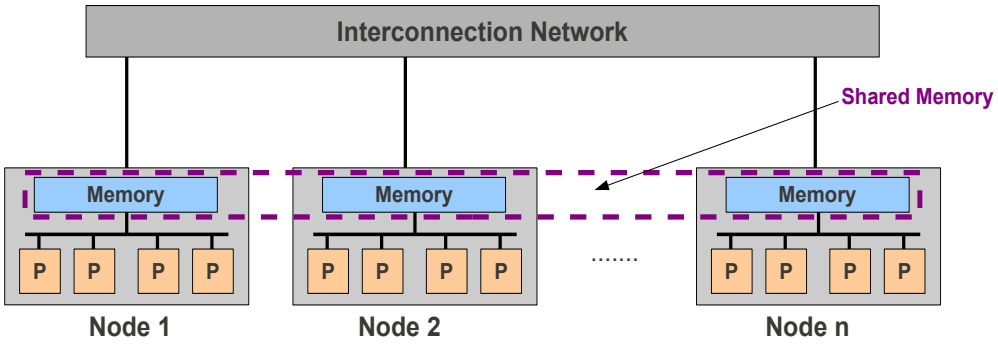


Figure 2.2: The NUMA Architecture.

A Non-Uniform Memory Access platform is a multiprocessor system in which the processing elements are served by multiple memory banks, physically distributed through the platform. Although the memory is physically distributed, it is seen by the machine processing units as a single shared memory. In these machines, the time spent to access data is conditioned by the distance between the processor and the memory bank in which the data is physically allocated [Carissimi 1999]. NUMA architectures are generally designed with cache memories, in order to reduce the memory accesses penalties. Due to this, some support to ensure the cache coherence for processing units are implemented in the current NUMA platforms, leading to cache-coherent NUMA platforms (ccNUMA). One of the advantages of NUMA architecture is that it combines a good memory scalability with an easy programming characteristic. In these machines, an efficient and specialized interconnection network provides support to the high number of processing units and very large memories. Since memory is seen as a global shared one, programmers can use shared memory programming models to develop parallel applications on these machines.

Figures 2.2 and 2.3 show schemata representing two NUMA machines. The one represented in Figure 2.2 reports the classical NUMA machines of the 80's whereas Figure 2.3 depicts a current NUMA machine with multi-core chips. We can observe in both figures that the NUMA machines are organized in multiple nodes connected by an interconnection network. Each node is generally composed of several processing units (mono-core or multi-core) and memory banks. In later sections, we explain the architectural differences between these two examples.

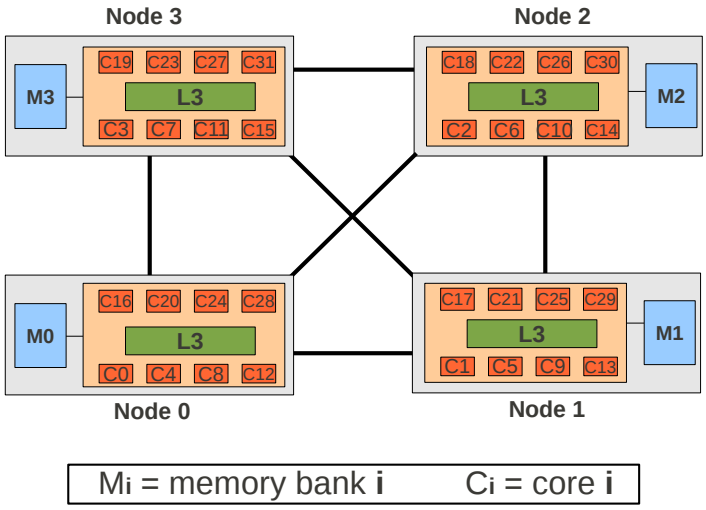


Figure 2.3: The Current NUMA Platform.

Another shared memory architecture that has hierarchical characteristics is the cache-only memory platforms. In this architecture the main memories are replaced by huge cache memories. Similar to NUMA architectures, this architecture also presents different memory access costs. However, differently from NUMA archi-

tectures, in the COMA platforms data is not associated with a node. In every access to some memory range, this range can be replicated or migrated to satisfy the request [Dahlgren 1999]. Due to this and to cache performance, COMA architectures generally have shorter memory latencies than NUMA machines. However, this architecture demands more specialized and complex hardwares to manage data coherence.

The advent of many-core machines has generated the need of more efficient cache subsystem architectures such as the *NUCA* ones. Non-uniform cache architectures were first introduced in [Kim 2002]. In these architectures the cache memory is larger than on non-NUCA architectures and it is split in several banks [Freitas 2009]. These banks are managed by memory controllers (MC), which gives the architecture different latencies to access the shared cache memory. NUCA architectures are very similar to NUMA ones, presenting non-uniform access to some cache memory level. Figure 2.4 shows a schema that depicts a NUCA processor architecture. The processor cores share a last level cache that is implemented with the NUCA design. Several banks composes the shared cache memory (16 banks in this example) that is controlled by some memory controllers. Each bank has a different access latency for each core of the processor, depending on the distance between the bank and the core.

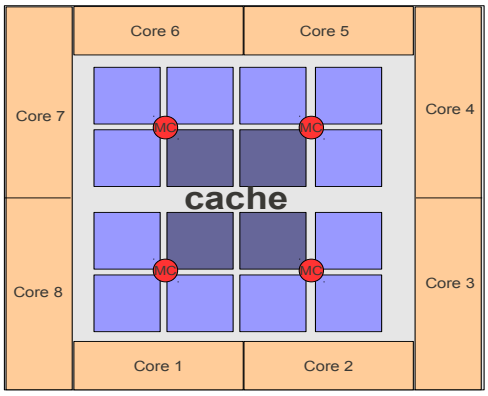


Figure 2.4: A NUCA Multi-core Processor Architecture.

In this thesis, we are specially interested in hierarchical shared memory architectures that presents non-uniform memory access costs, the NUMA platforms. This machines are becoming a trend in the High Performance Computing (HPC) centers. Particularly, the multi-core machines with NUMA design are used as building blocks of the current HPC clusters. These multi-core NUMA platforms present complex topologies and hierarchical memory subsystem, which must be well exploited in order to have scalable performances. In the next sections, we present the evolution of multiprocessors that have led to the proposal of NUMA architectures.

2.2 Evolution of Shared Memory Multiprocessors

As mention in the previous section, the shared memory multiprocessors with UMA design present some drawbacks related to the interconnection network. In this context, UMA multiprocessors are limited to a number of processing units, reducing its scalability and processing power. In order to provide more scalability to multiprocessors, by the end of the 80's research groups proposed the first multiprocessors with NUMA design. In this section, we present a deeper description of this architecture and its components. We also discuss the architectural evolution concerning the shared memory multiprocessors.

2.2.1 More Scalability with NUMA Architectures

The architectural design of shared memory machines with UMA characteristics leads to poor scalability. Because of this, some issues such as the memory wall [McKee 2004] problem, the limitation in power processing and memory capacity appears in these architectures [Dupros 2009]. The memory wall problem occurs because processing units operate faster on data than memory can provide data [Wulf 1995]. Memory is usually slower than processing units, due to its technology and also due to the single interconnection network of UMA shared memory machines.

One possible solution for all these problems is the use of non-uniform memory access architectures to build more scalable shared memory platforms. The non-uniform memory access design allows shared memory machines to distribute the access of processing units in a distributed shared memory, providing more bandwidth for each processing unit. In this case, the use of specialized technologies for the interconnection network and cache memory hierarchy are included in the machine design. These technologies allow the multiprocessors to have much more memory capacity, a shorter latency to access memory and a larger number of processing units.

In the end of the 80's, a number of research groups have started to work on prototypes of multiprocessors that rely on NUMA characteristics [Lenoski 1993, Agarwal 1995, Falsafi 1997]. The DASH (Directory Architecture for Shared memory) project was the first initiative to build a NUMA machine [Lenoski 1993]. The project goal was to design large-scale single-address-space machines with some cache coherence support. To do so, they have distributed the global shared memory through different nodes to provide scalable bandwidths. In this architecture, each processor has its own cache memory to reduce memory access costs. Due to this, it has also included a directory protocol to guarantee cache coherence among processors. The main contribution of this project is the implementation of a novel architecture that adds the efficiency and scalability of Massive Parallel Processing (MPP) to the easy programmability of Symmetric Multiprocessors (SMP).

Figure 2.5 shows a schema of this architecture with its hardware components. The figure reports a DASH architecture with four nodes interconnected by a mesh network. The global memory is physically distributed among the four nodes and a

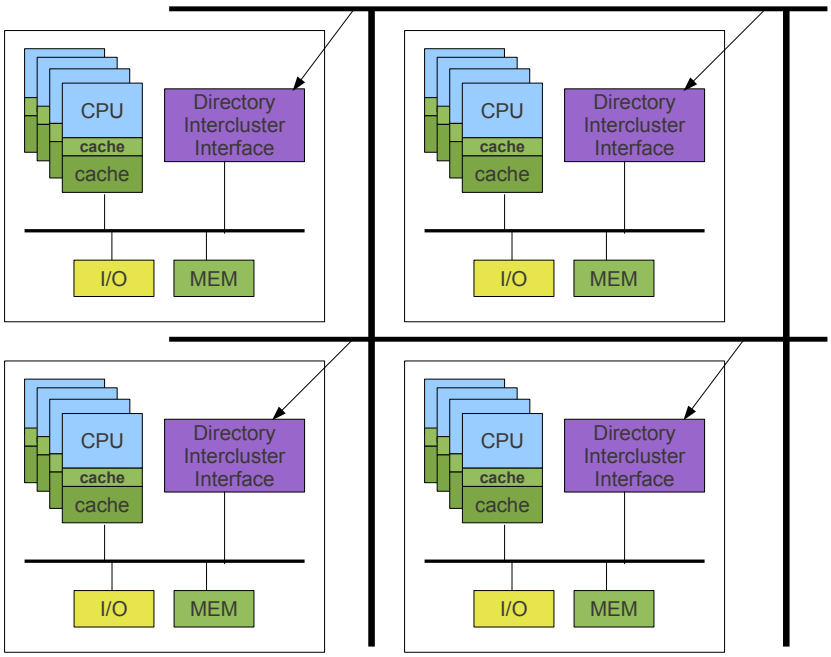


Figure 2.5: The DASH Architecture [Lenoski 1993].

specialized hardware provides the shared memory. Cache coherence is ensured by the directory cluster interface hardware.Additionally, each CPU has its own private cache memories to reduce communication costs.

The advent of the DASH architecture brought the concept of remote and local accesses, that was before related to clusters of shared memory machines. In this way, a local access is performed when a processing unit access the memory that resides in the same node it belongs to. Contrary to this, the remote access occurs when a processing unit request data that is allocated on a node which is different of the one the processing unit belongs to. Figure 2.6 illustrates both local (red arrows) and remote accesses (blue arrays).

After the DASH, several other NUMA architectures were proposed in the 90 decade. Some examples are the KSR1 [Singh 1993], the Cray T3D [Cray 1993], the Reactive NUMA [Falsafi 1997] and the Alewife [Agarwal 1995]. The difference between all these architectures are the design of main memory access (DRAM or cache-only memory), the support or not to cache coherence and the interconnection topology. After the first explosion of NUMA architectures, manufacturers such as SGI and IBM have started to build large-scale multiprocessors with NUMA features. Nowadays, SGI and IBM continue to use the concept of NUMA on the design of machines for different fields such as web servers and high performance computing servers.

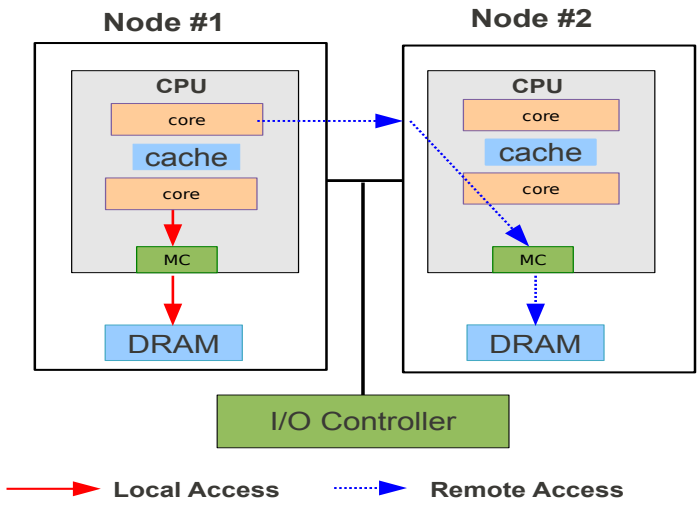


Figure 2.6: Local and Remote access inside the NUMA Platform.

2.2.2 Mono-core to Multi-core Platforms

In the last decade, the main challenge for computer science researches in the computer architecture domain has been to keep up the well-known Moore's law [Moore 2000]. This law states that processor power duplicates every 18 months. However, by the end of 90's computer architecture researchers already had problems in confirming this law in the design of mono-core processors.

A mono-core processor is characterized by a single processing unit (core) that is responsible for computing all the instructions, Figure 2.7 (a). This single processing unit generally has an arithmetic logic unit and some memory components to store data. In the mono-core processor architecture, in order to increase its frequency new design was required. Due to the demand for more powerful processors, researchers have made use of instruction-level parallelism and thread level parallelism mechanisms to enhance processors performance [Patterson 2009]. The instruction-level parallelism allows the execution of multiple instructions at the same time. In order to support the instruction-level parallelism, mono-core processors implement pipeline and multiple arithmetic units techniques. The pipeline technique splits the execution of an instruction in multiple steps, allowing some of them to be executed in parallel [Ramamoorthy 1977]. The multiple arithmetic units technique allows the execution of several operations at the same time. The thread level parallelism mechanism supports the execution of multiple threads in the same mono-core processor [Marr 2002]. This mechanism is possible due to the replication of some components of the processor. However, even these mechanisms did not make possible the necessary performance improvement for the parallel machines.

Since mono-core processors have been used to build large scale SMP machines, researchers groups have worked to improve their power by including the multi-core

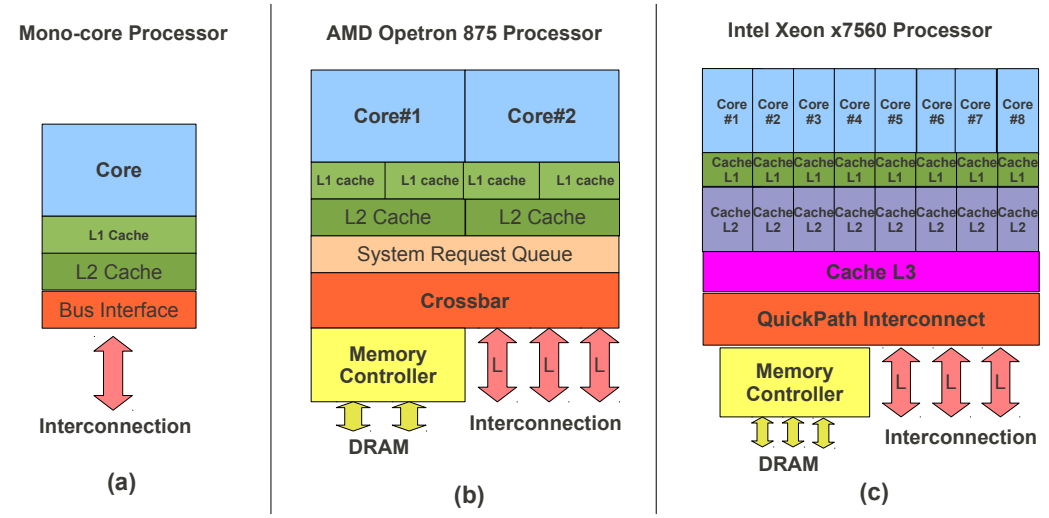


Figure 2.7: The Evolution: Mono-core to Multi-core Chips.

design inside them. A multi-core processor is composed of two or more independent cores that share some hardware components, Figure 2.7 (b) and (c). The main concept behind the multi-core architecture is the use of processing units with shorter frequencies that work together to accomplish a job. In this way, multi-core processors can do more work than mono-core processors, because of their implicit parallelism. Additionally, the distance between the processing units and memories is smaller in multi-core processors. Consequently, latency to perform communication can be shorter, improving the overall performance of the processor.

The first multi-core processor was built in 2001 by IBM, the Dual Core POWER4+. Even tough multi-core chips are more expensive than mono-core ones, several manufactures have started to design and build their own multi-core processors, e.g. AMD, Intel and SUN. Figures 2.7 (b) and (c) show a schema of two different multi-core architectures designed by AMD and Intel. We can observe that multi-core platforms feature complex topologies with several cores in a single chip. In this architecture, hierarchical cache memory subsystems with several levels are common resources. Still, multi-core processors can also make use of the instruction-level parallelism and the thread level parallelism mechanisms.

Multi-core technology is expected to solve the problem of achieving higher performances on shared memory machines. Because of that, the number of cores per processor will continue to increase, resulting in many-core architectures with hundreds of cores. Therefore, the software level such as operating systems, runtime systems and applications must be adapted to efficiently use the multi-core features. Applications must be aware of the shared caches and how cores are assembled inside the processor chip. For instance, managing thread and data placement efficiently can be crucial for the application performance in multi-core machines. Chapter 3 details the software support to deal with such issues on shared memory machines

based on multi-core processors.

2.2.3 Multi-core Platforms With NUMA Characteristics

The multi-core architecture is a trend in different areas of computer science, specially in HPC. It addresses important issues inside processors such as the instruction level parallelism within a chip and the power wall problem [Liu 2009]. These problems are reduced by multi-core design because it provides multiple levels of cache memories, multiples pipelines and less consuming processing units. However, other problems arise because of the multi-core characteristics. For instance, the memory wall problem at main memory level is an important one.

In the context of hierarchical shared memory platforms, multi-core processors have been used as building components. Using multi-core processors, computer architecture engineers built powerful shared memory machines with tens or even hundreds of cores. The problem appears when all these cores request some data that is not present in any of the cache memories. Therefore, the data request is answered by the global shared memory, which can consume several CPU cycles because of the single bus. During these CPU cycles cores are idle waiting for data.

In order to reduce such a penalty, processors manufactures have brought to multi-core architectures the concept of NUMA. The non-uniform memory access is implemented in multi-cores in a similar way as the first NUMA platforms of the 80's and 90's. Some hardware support is included inside the machine to handle the physical distributed shared memory. This hardware can be implemented as a specialized hardware that control all access to memory or as an integrated memory controller (MC) in the processor chip. The latter one has been more adopted by manufactures because it reduces the centralized management of all memory requests [Awasthi 2010].

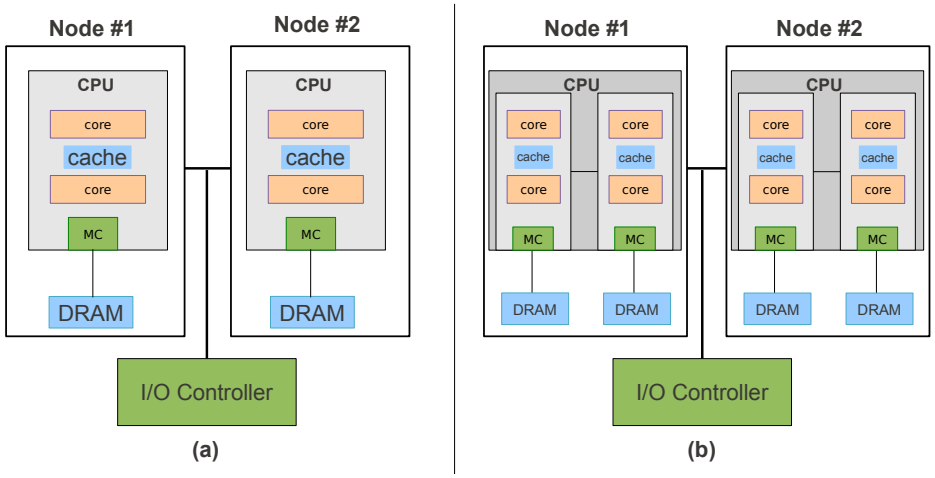


Figure 2.8: Multi-core NUMA Platforms: (a) Single Memory Controller (b) Multiple Memory Controller.

Several multi-core architectures such as Nehalem from Intel and Opteron from AMD feature a NUMA design. Figure 2.8 (a) shows a schema of a parallel machine based in multi-core processors, where each processor is composed of two cores, a shared cache memory and a memory controller. In this architecture, cores in the same processor communicates with each other using the shared cache. This design reduces communication costs between cores and also memory access costs to get some data. The memory controller inside the processor is responsible for the management of all requests to get data from memory. In the architecture illustrated in this figure, each processor has its own memory controller. However, with the increase of the number of cores per processor, other designs are necessary.

Since memory controllers are responsible for handling all memory requests, it can become a point of bottleneck. In order to avoid the contention on the memory controller, manufactures have started to integrate multiple memory controllers per processor. Each memory controller is responsible to answer requests from a restricted set of cores, reducing the contention on memory controllers queues [Wentzlaff 2007]. Figure 2.8 (b) depicts a schema of an architecture where multiple memory controllers per processor are used. This architecture also presents NUMA characteristics and it is more hierarchical than the ones with single memory controllers per processor. Therefore, it demands an efficient data placement and control of all requests to memory controllers.

Similarly to the first NUMA machines, the current NUMA multi-core platforms also have their design cache coherence protocols to ensure data consistency. The main difference is in the algorithm that ensures the cache coherence. Other concepts that are also present in the modern NUMA platforms are the local access and remote access. Here, the main difference is that current NUMA platforms tend to present smaller local and remote access costs thanks to the new interconnection network technologies.

In order to provide an idea of the difference between current NUMA machines and the machines proposed in the 90's, Table 2.1 shows two memory performance metrics for three NUMA machines. The performance metrics presented in the table are the memory bandwidth and the ratio between remote and local latency. We can observe that current NUMA machines (AMD 875 and Intel X7560) have a smaller ratio between remote and local latencies than the DASH machine (extracted from [Lenoski 1993]). Additionally, memory bandwidth of the current NUMA machines is higher than the ones of the DASH platform.

Table 2.1: NUMA Platforms Characteristics.

Characteristic	**DASH**	**AMD 875**	**Intel X7560**
Memory bandwidth	64 MB/s	9.77 GB/s	35.54 GB/s
Ratio Remote vs Local Access	4.55	1.5	2.6

Although, the current NUMA multi-core machines have shorter latencies and higher memory bandwidths, these characteristics do not eliminate the NUMA penal-

ties. For instance, in the same table we can observe that on an Intel based machine the ratio between remote and local access on shared memory can be up to 2.6, which means that a wrong data/thread placement generates a memory access cost of 2.6 times greater than the local access costs. Furthermore, in current NUMA machines one of the main challenges is to deal with the complex topology and hierarchy of the machine. In the chapter 3, these challenges are introduced and described.

2.3 Memory Subsystem Hardware for Hierarchical Architectures

One of the main advantages of NUMA machines is that programmers can develop parallel applications using the shared memory programming model. This is possible because of the specialized hardware that provides a single address space at the memory level. However, as discussed in previous sections, the implementation of the shared memory with NUMA characteristics demands a interconnection network and cache coherence protocols. These two components are responsible for the NUMA penalties, since they deal with data transfers and update. In this section, we introduce and describe some of the interconnection network and cache coherence protocols used in the design of NUMA machines.

2.3.1 Connections between Processors and Memory

In order to build NUMA machines, it is necessary to have an interface between the processor and the other devices (e.g. memory and peripheral devices). This interface is the interconnection network that makes possible the exchange of data among the platform components. Generally, each processor manufacturer implements its own interface architecture to have the best performance for its architecture. In the context of NUMA platforms several interconnection network have been designed by different manufactures [Intel 2011b, AMD 2011a, SGI 2011].

The Front-Side Bus (FSB) is an interconnection architecture developed to connect processors to the memory and peripheral devices. It has been used by several processor manufactures on its shared memory machines since the 90's. One of the biggest disadvantages of FSB is that each implementation is machine specific. Therefore, each processor manufacture design its own FSB architecture. In the case of Intel, the FSB first implementation relies on a single shared bus that connect all processors. Over the years this architecture has been improved to provide more bandwidth to processors. The current implementation of FSB defines a point-to-point interconnection, which enhances the available bandwidth for each processor.

The advent of memory controllers integrated in the processor chips and the demand for more bandwidth, forced the replacement of the FSB to new interconnects such as the Quick Path Interconnect [Intel 2011a] and the HyperTransport Interconnect [AMD 2011a]. Most of NUMA multi-core machines based on Intel processors use the new interface named QuickPath Interconnect (QPI). The architecture design

for this interconnect relies on point-to-point link that allows data exchange in multiple lanes. Additionally, this interconnect splits packets to send them in parallel. Thanks to these features the QPI has the capacity to provide better bandwidths to the current multi-core machines. The QPI also implements cache-coherence protocol that guarantees that data is coherent for all levels of the memory subsystem. For instance, QPI can be up to 50% better than FSB interconnect [Intel 2011a].

The competitor to QPI is the HyperTransport (HT) interconnect, which is an open specification that has been adopted by AMD. The first version of HT was introduced in 2001. After that, most of NUMA machines based on AMD processors have been implemented with this technology. Similar to QPI, HT also performs data transfers in parallel, allowing more bandwidth. Compared to FSB, the HT version 3.0 can be 20% better in terms of data transfer performance [AMD 2011a]. The main difference between QPI and HT is the cache coherence protocol supported by them. QPI uses the MESIF protocol, whereas HT uses the MOESI protocol. In the next section, we provide more details about both of them.

2.3.2 Cache Coherence Protocol for NUMA Platforms

On a NUMA platform, the cache coherence protocol is responsible for keeping all shared data consistent across its different processing units. The cache coherence protocol specifies actions that must be executed when read and write operations are performed on some data (cache line). Several protocols have been proposed in order to efficiently implement these actions. Concerning the NUMA platforms, there are different cache coherence protocols implementations such as MESI, MOESI, and MESIF [Papamarcos 1984, AMD 2011b, Molka 2009]. The first letter of each possible state in each protocol is used to name it. These protocols have been used on current NUMA machines based on Intel and AMD processors.

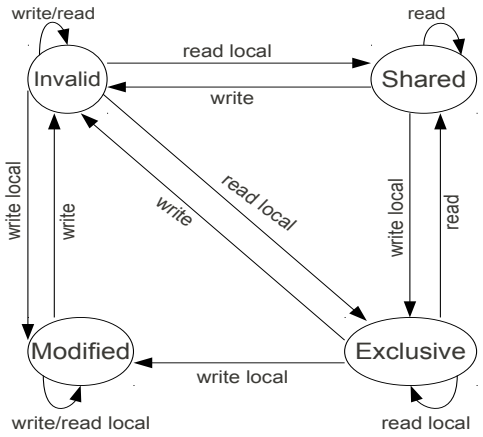


Figure 2.9: MESI State Transitions [Technion 2011].

The MESI (Modified, Exclusive, Shared and Invalid) protocol has been proposed

at the University of Illinois [Papamarcos 1984]. This protocol has four states and it is generally used with the FSB interconnect. The possible states are: (i) modified, the cache line has been modified but not yet updated in main memory; (ii) exclusive, the cache line is updated in one cache and matches with main memory; (iii) shared, the cache line can be used by other processor; (iv) invalid, cache line is invalidated and can not be used. From a state, the protocol defines which states can be reach for a cache line.

Figure 2.9 shows the state transitions for MESI protocol. Basically, what this diagram illustrates is how read and write operations are performed. For a read operation from a processor, messages are sent to request data if the processor does not have data in its cache (the state invalid). If the processor has data, in the case of any other state, it performs the read in a local way. For write operation on data, processors have to go to the modified mode. However, if the processor does not have the data, it has to request it to other processors. In the case of one of the processors have a copy of the data that will be written, it has to go to the exclusive mode to invalidate any copy and then it goes to the modified mode. Due to its design, this protocol presents low bandwidth to communicate to other processors and it demands that a modified cache line must be updated in main memory. Furthermore, in this protocol on every read and write operations it has to check if data was modified.

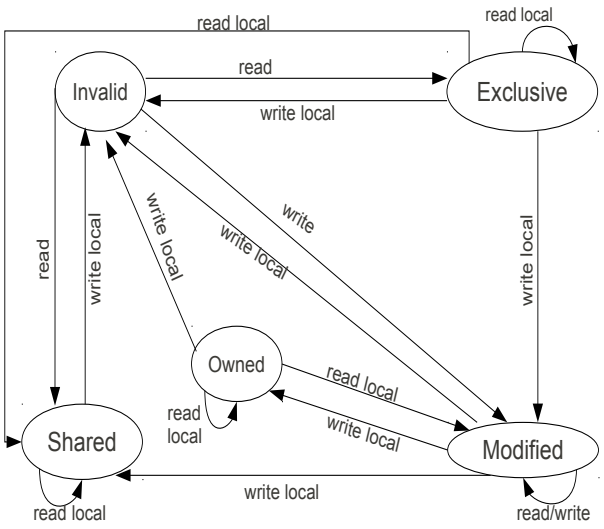


Figure 2.10: MOESI State Transitions - Extract from [AMD 2011b].

In order to reduce the number of messages and hops need to perform a read/write on MESI, some modifications have been proposed to this protocol. The inclusion of a new state to reduce the overhead of the MESI protocol has originated the MOESI and MESIF protocol. The MOESI protocol includes the owner state (O), whereas the MESIF includes the forward state (F). The MOESI has been used in the AMD Opteron processors and the MESIF in the Intel Nehalem processors.

The inclusion of owner state in MOESI protocol creates a state that represents a data that is modified and shared. When a processor asks data for read and other processor has it in the owned state, this later can supply the request. Therefore, this new state avoids the need to write a cache line back to main memory when it has been recently modified by a processor. This protocol is generally more performant than the MESI one. Figure 2.10 shows the state transitions for MOESI protocol. We can observe in the figure that both owned and modified states obligate a cache line invalidation in the case of a write operation.

In the case of MESIF, the inclusion of the forward state avoids multiple answers to a read miss. In every read request from a processor, a broadcast is performed to ask other processor for data. However, contrary to MESI protocol only one processor will answer the request, the one that holds data in the forward state. This strategy reduces the amount of messages necessary to maintain the coherence between caches, enhancing the overall performance of the machine. Figure 2.11 shows the state transitions for MESIF protocol.

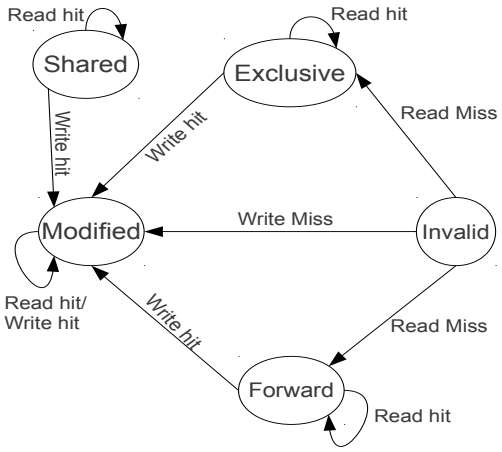


Figure 2.11: MESIF State Transitions.

The main difference between these three cache coherence protocols is the number of hops and messages needed to get some data on a read or write operation. These two parameters impact directly on the memory performance of the NUMA machine and consequently, on the application performance. A number of works have evaluated such differences on different multi-core platforms, their analysis are presented in [Molka 2009, Hackenberg 2009].

2.4 Conclusions

Current large-scale shared memory machines rely on multi-core processors with non-uniform memory access (NUMA) characteristics. Due to this design, shared memory machines are becoming more complex with a hierarchical topology and

memory subsystem. NUMA multi-core machines feature multiple cache levels, specialized interconnection networks, several cores and a distributed shared memory. Since these parallel platforms are becoming a trend in scientific High Performance Computing (HPC), it is important to design some support to efficiently use their resources. The architectural characteristics of such machines increase the complexity of extracting scalable performances from these machines. Therefore, the main challenge of NUMA multi-core machines is ensure efficient use of the machine resources.

In this chapter, we observed that several architectural design choices might have an important impact on the overall performance of applications executed in the NUMA multi-core machines. For instance, the remote latencies, the memory bandwidth and the cache coherence protocol overhead can have a significant role in the system performance. In order to reduce such NUMA penalties, it is important to efficiently manage thread and data placement on the NUMA multi-core machines. In this context, improving memory affinity between processing units and data becomes necessary on these platforms.

Memory affinity ensures that thread and data will be efficiently mapped over the machine in order to optimize latency and bandwidth. Thus, on NUMA platforms, parallel programming languages and libraries must be aware of memory affinity in order to extract scalable performances from the hardware. In the next chapter, we discuss the current NUMA support in the software level in hierarchical shared memory machines.

Chapter 3

Software Issues of Memory Affinity Management on Hierarchical Multi-core Machines

It was shown in the previous chapter that the current shared memory machines rely on a complex architecture design. This complexity is mainly attributed to the non-uniform memory access characteristics of its memory subsystem and the core topology present in the processors. Such characteristics can lead to poor performances if the software level is not prepared to deal with them efficiently. In this context, enhancing memory affinity between threads and data in the software level for parallel applications becomes crucial. Memory affinity is enhanced when thread and data are mapped over the machine in a such way that it reduces latency and increases bandwidth. In this chapter, we discuss the software issues related to the management of memory affinity on multi-core machines with NUMA design. The chapter goes through the state of art of the software level, highlighting its drawbacks on memory affinity support for current machines. We start by introducing some parallel programming languages and systems used in the development of parallel applications for shared memory machines. We then discuss the false sharing problem in the context of NUMA machines. Forward, we present a case study of the penalties that the NUMA characteristics can incur on the performance of parallel applications. We depict the approaches and software support to deal with memory affinity management on shared memory machines. Finally, we present a conclusion on the state of the art in software solutions for memory affinity. This discussion leads to the motivation for the contributions of this thesis.

3.1 Parallel Programming on Shared Memory Platforms

In order to obtain scalable performances from parallel platforms in high performance computing, it is mandatory to use efficient parallel programming languages and systems for the application development. Several parallel programming environments have been designed over the years. Nowadays, we count on a large set of parallel programming languages and systems to implement efficient parallel applications for different parallel architectures. Some examples are MPI (Message Passing Interface) [Snir 2010], PM^2 [Namyst 1997], Charm++ [Kalé 1993, PPL-Charm++ 2011], OpenMP [OpenMP 2011], HPF (High Performance Fortran)

```
#define ITERS 1000
#define MAXGRID 2048

double grid1[MAXGRID][MAXGRID], grid2[MAXGRID][MAXGRID];
int id;

 for (iters = 1; iters <= ITERS; iters++) {
  #pragma omp parallel for private(j) schedule (static){
    id = omp_get_thread_num();
    for (i = 1; i < MAXGRID - 1; i++)
      for (j = 1; j < MAXGRID -1; j++)
        grid2[i][j] = (grid1[i-1][j] + grid1[i+1][j] +
                       grid1[i][j-1] + grid1[i][j+1]) * 0.25;
  }
 }
```

Figure 3.1: OpenMP Example.

[Richardson 1996], UPC (Unified Parallel C) [UPC 2011], Software Transactional Memory [Larus 2006] and CUDA [NVIDIA 2010].

The examples cited above can be classified into message passing or shared memory programming models. In the message passing programming model, the application computation is executed in parallel by processes. Every time a process needs to communicate with another process (to exchange data for example), they send and receive messages from each other. Generally, the message passing model scales better, but it demands a more complex coding effort to implement an application. Shared memory programming model relies on the concept of a shared global memory space that is used by processes and threads to communicate. This programming model is usually less scalable than the message passing one, but simpler and easier to be used by parallel application developers [Grama 2003].

Considering multi-core platforms with NUMA design, the most suited programming model is the shared memory one, since the machine is based on a shared memory model. There are several popular parallel programming languages and systems that support the shared programming model. In this thesis, we will focus our state of the art on OpenMP, HPF, UPC, OpenSkel [Góes 2010b] and Charm++. We have focused on such languages and interfaces because they represent an effort on HPC to reduce the complexity of programming parallel applications while attaining scalable performance. These languages and interfaces try to hide technical details from the developer when implementing a parallel application.

OpenMP is an API (Application Programming Interface) that supports the shared memory programming model on multiprocessors machines [OpenMP 2011]. By the end of the 90's decade, a consortium between companies and universities proposed OpenMP with the objective of being a simpler and easier way to program parallel applications. The consortium defined a specification for OpenMP that describes a set of directives, functions, and environment variables for the API.

```
PROGRAM MATRICES
integer N, I, J
real grid1(N,N)
real grid2(N,N)

%!hpf$     PROCESSORS P(4)
%!hpf$ distribute grid1(block,block)
%!hpf$ distribute grid2(block,block)

  DO ITERS=1,MAXITERS
        DO I=1,N FORALL (J=1:N)
           grid2(I,J) = (grid1(I-1,J) + grid1(I+1,J)+
                 grid1(I,J-1) + grid1(I,J+1)) * 0.25;
        END DO
   END DO
END PROGRAM
```

Figure 3.2: HPF Example.

OpenMP has been implemented to C, C++ and different versions of Fortran, all the while leaving implementation details and choices up to each compiler. The only rule is that OpenMP implementations must follow the specification proposed by the consortium [OpenMP 2011]. In this way, programmers have a portable API to develop their applications on. The OpenMP constructions allow developers to control the parallelism, the synchronization, the work-sharing and the mutual exclusion for an application. The parallelism on OpenMP relies mostly on the parallelization of loops, indicating which set of instructions must be executed in parallel. Figure 3.1 shows an example of an OpenMP code in C. In this example, we present some of the OpenMP directives (*pragma omp parallel for* to parallelize the **for** loop), functions (*omp_get_thread_num()* to get the OpenMP id of a thread) and clauses (*private* for data sharing attribute and *schedule* for work distribution).

HPF is an extension to Fortran 90 with parallel programming support proposed by the HPFF (High Performance Fortran Forum) in 1993 [Richardson 1996]. HPF relies on the data parallel model and it has special support for data (array) distribution over the machine. Data distribution is performed considering the distribution strategy chosen by the user at the data declaration. This feature allows HPF to be efficiently implemented on shared and distributed memory architectures. However, some definitions of HPF are complex to be integrated inside a compiler and vendors have abandoned their efforts to implement HPF. Most HPF users adopted OpenMP as a substitute for HPF. Figure 3.2 reports an example of a HPF code. In this example, we present some of the HPF directives: *PROCESSORS P(4)* that defines how many processors will be used for work distribution, *distribute grid1(block,block)* that defines how the elements of the grid1 matrix will be distributed for the worker threads and *FORALL* that allows any order execution of its content.

UPC is an extension of ANSI C programming language with parallel programming support proposed by the UPC consortium in 1999 [Carlson 1999, Coarfa 2005].

```
#define ITERS 1000
#define MAXGRID 2048
#define CHUNK MAXGRID/NTHREADS

shared [CHUNK] double grid1[MAXGRID][MAXGRID],
            grid2[MAXGRID][MAXGRID];

 for (iters = 1; iters <= ITERS; iters++) {

  upc_forall (i = 1; i < MAXGRID; i++;i) {
      for (j = 1; j < MAXGRID; j++)
        grid2[i][j] = (grid1[i-1][j] + grid1[i+1][j] +
                       grid1[i][j-1] + grid1[i][j+1]) * 0.25;

      if(MYTHREAD == 0)
         printf("\n master thread doing work");
  }
}
```

Figure 3.3: UPC Example.

The main objective of UPC is to provide a parallel programming support based on the partitioned global address space programming model. In this model, the language abstracts from the programmer the notion of remote and local memory accesses. Using special notations, developers can access data as if they are programming for a shared memory machine. Nowadays, several implementations of UPC have been released in compilers from HP, SGI, Sun and Cray. UPC introduces new concepts for data sharing, thread affinity and work distribution. Figure 3.3 shows an example of a UPC code. In this example, we present some of the UPC keywords: *shared [CHUNK]* that defines how shared data will be distributed (round robin way in chunks of CHUNK) for the workers, *upc_forall* that allows users to parallelize a sequence of instructions and *MYTHREAD* that allows user to get the UPC id for a thread. Differently of the standard C **for**, the *upc_forall* has a fourth parameter that specifies the affinity between threads and data. In the example, the affinity between threads and data is defined by i, which defines the arrays rows. Therefore, affinity is performed by rows in a round robin way over threads. Additionally to these features, UPC also defines some support for data coherence (sequential and relaxed [Carlson 1999]) and parallel I/O.

OpenSkel is a transactional skeleton framework developed in the University of Edinburgh [Góes 2010b, Góes 2010a]. It combines parallel skeletons with transactional memory software leading to a transactional skeleton programming model. This new model enjoys the benefits of both approaches, simplifying parallel programming and improving performance. In this sense, transactional applications can be tuned transparently and automatically by the skeleton framework. OpenSkel relies in a C based API that allows programmers to develop parallel application in

```
int main(int argc, char **args) {
 //shared data must be allocated using this data type
  oskel_w1_shared_t global;

  global = malloc(sizeof(oskel_w1_shared_t));
  global->data = malloc(sizeof(args[1]));

  //allocate the pointer to the worklist
  oskel_w1_t *oskelPtr=oskel_w1_alloc(sizeof(oskel_w1_t),&global);

  //include a work unit in the worklist
  while(!feof(file))
     oskel_wl_addWorkUnit(oskelPtr,read(file));

  //starts the execution - create the team of threads
  oskel_wl_run(oskelPtr,global);

  //work finished - free the varibles
  oskel_wl_free(oskelPtr);
}
```

Figure 3.4: OpenSkel Example.

a simple way. The API provides data types and functions to define the application data, work units that composes a computation and the start of computation. In Figure 3.4, we present a worklist example of an OpenSkel application. In this example, the global shared data is allocated and the OpenSkel pointer to the worklist *oskelPtr* defined. In OpenSkel API all shared data must be encapsulated in the data structure *oskel_w1_shared_t*. Work units are added to the OpenSkel worklist with the routine *oskel_wl_addWorkUnit(oskelPtr,input)* and processed using the routine *oskel_wl_run(oskelPtr,global)*.

Charm++ is a C++ parallel programming system that was proposed in 1993 in the Parallel Programming Laboratory at the University of Illinois. It aims at providing a parallel programing model that abstracts architecture characteristics from the developer, improving programmer productivity. Charm++ introduces the idea of parallel objects to encapsulate the computation and communication for an application. Such objects are automatically mapped to processors by Charm++ runtime and they communicate with each other using a message driven model [Kalé 1993, Kalé 2009a]. Figure 3.5 shows an example of a Charm++ application that was extract from Charm++ examples [PPL-Charm++ 2011]. In this example, an array (*arr*) of nElements is created and a message is passed along the array from the element 0 to N. The routine *CProxy_Hello::ckNew(nElements)* creates the array, whereas the routine *SayHi(int hiNo)* is used to pass the message.

The advent of multi-core NUMA machines has increased even more the need of efficient software for high performance computing (HPC). Most of the languages and interfaces mentioned in the previous paragraphs do not have any support for the current NUMA multi-core machines. Furthermore, in the ones with some affinity

```
#include "hello.decl.h"

/*readonly*/ CProxy_Main mainProxy;
/*readonly*/ int nElements;

/*mainchare*/
class Main : public Chare { public:
  Main(CkArgMsg* m) {
    //Process command-line arguments
    if(m->argc >1 )
       nElements=atoi(m->argv[1]);

    //Start the computation
    mainProxy = thisProxy;
    //create the array of objects
    CProxy_Hello arr = CProxy_Hello::ckNew(nElements);
    //starts to transmit the message
    //message is the number 17
    arr[0].SayHi(17);
  };

  void done(void)  {
    CkPrintf("All done\n"); CkExit();  };
};

/*array [1D]*/
class Hello : public CBase_Hello {
public:
  Hello() {
    CkPrintf("Hello %d created\n",thisIndex); }

  Hello(CkMigrateMessage *m) {}

  void SayHi(int hiNo)  {
    CkPrintf("Hi[%d] from element %d\n",hiNo,thisIndex);
    //continue to send messages - next object
    if (thisIndex < nElements-1)
      thisProxy[thisIndex+1].SayHi(hiNo+1);
    else //message has arrived
      mainProxy.done();
  }
};
```

Figure 3.5: Charm++ Hello Word Example - Extracted from Charm++ Examples.

support (e.g. UPC), programmers are obliged to explicitly manage all the affinity, providing some hints in the application source code. We observed in chapter 2 that the multi-core NUMA platforms demand from parallel applications an efficient use of the machine topology and interconnection network, in order to reduce memory access costs. In the next sections, we present the performance problems that NUMA machines can bring to parallel applications. We then present the efforts that research groups have done in order to get better performances from parallel applications in such machines.

3.2 False Sharing in NUMA Platforms

The development of parallel applications for shared memory machines can be simplified thanks to shared memory programming support available on some languages as presented in the previous section. However, developers must be aware of some performance issues that can arise because of the architectural characteristics of shared memory machines. One of these issues is the false sharing problem.

False sharing occurs when two or more processing units require different data within the same coherence block (i.e. cache line). In this situation more coherence operations are necessary to keep data correct for all processing units. This problem is presented on shared memory machines that have cache memories and implement a cache coherence protocol. This is a common factor for poor performance on such machines, because higher latency and lower bandwidth are caused by the coherence operations [Bolosky 1993].

An example of false sharing problem in a parallel code is presented in Figure 3.6. In this example, we assume that there are two threads running and that they share a cache memory. Additionally, the cache line holds one integer. Therefore, each time that Thread 0 writes to *var*, it will invalidate the cache block for the other thread. Even tough threads write to different logical positions, in the hardware these positions are in the same cache block producing the false sharing problem. Therefore, the average access time to get data for each thread is increased, degrading the application performance.

```
//global variables
int var,varOdd;

//executed by all threads void makeInc() {
   if (myId() == 0)
     var = var + 1; else
       varOdd = varOdd + 1;
 }
```

Figure 3.6: An Application with False Sharing Problem.

Particularly on NUMA platforms, this problem worsened because of the non-uniform memory access cost on these machines. Memory access costs are mainly

related to the cache coherence protocol operations and the interconnection network between NUMA nodes.

In order to deal with this problem, developers can make use of compiler support, automatic mechanisms and profiling/tracing tools that help them to identify and solve the false sharing point. Some techniques such as data padding, privatization of shared data, data allocation and thread placement can also be used by developers to reduce false sharing impact [Jeremiassen 1995, Berger 2000, Broquedis 2009].

3.3 A case study: NUMA Impact on Parallel Applications Performance

In this section, we report how the NUMA design can impact the overall performance of parallel applications on multi-core machines. To do so, we use two synthetic applications written in C/C++ language that represent different behaviors of a parallel application. For this study, we chose Charm++ and OpenMP as solution for code parallelization because of their simplicity of usage and difference in the programming model. Furthermore, similarly to other parallel languages and interfaces, none of them has any NUMA support.

We start with a simple example that shows the impact that a NUMA machine can cause on a parallel application. In order to demonstrate that, we present the performance difference of a benchmark written in Charm++ when executed in multi-core machines with UMA and NUMA design. The benchmark performs a Jacobi computation in a two-dimensional matrix and the complete code is presented in [Jacobi 2011]. In this benchmark, to compute a point of the matrix, a task needs to use the south, north, west and east points of the matrix. This means that some communication between tasks must be performed in order to get the necessary data.

Table 3.1: Speedups for the Charm++ Jacobi Benchmark.

Number of Threads	UMA	NUMA
4	2.26	2.31
8	6.12	6.15
16	7.01	6.10
24	8.77	7.01
32	-	7.02

Table 3.1 shows the speedup for the Jacobi benchmark when executed in two multi-core machines, the first one with UMA design and the second one with NUMA design. The UMA machine has 24 cores and all cores have similar memory access latencies to get data. In contrast, the NUMA machine has 32 cores, grouped in 8 cores per NUMA node with different memory access latencies to get data. We can observe in Table 3.1 that while the speedups in the UMA machine presents some

scalability, the ones obtained in the NUMA machine do not.

Considering the Jacobi benchmark characteristics, such NUMA impact comes from the communication step needed to get the four neighbors points. In the case of the NUMA machine, this communication is more expensive due to the remote accesses that must be performed to get data on the NUMA nodes. It is important to mention that both machines have similar processor frequencies and a shared last level cache memory. We can then conclude that the performance penalties on the NUMA machine comes from its non-uniform memory access design.

```
//to reduce cache influence - large vector
#define N 2000000
#define NTIMES 100

  double          c[N],a[N];

  for (k=0; k<NTIMES; k++) {
    #pragma omp parallel for
        for (j=0; j<N; j++)
            c[j] = a[j];
  }
```

Figure 3.7: Synthetic Application Snippet.

In our second case study, we present a simple example that demonstrates how data initialization on NUMA machines can have an important impact on an OpenMP application performance. Figure 3.7 depicts the snippet of a synthetic code that copies a vector onto another one. In this code, each thread computes a chunk of the arrays. The chunk size in this example is N/number of threads. We use for this case study two types of data initialization, the master thread and the team thread. In the first one, all data is initialized only by the master thread, whereas in the second one each worker thread initializes its own data using the OpenMP construction *parallel for* (Figure 3.8).

The NUMA machine used in this study runs Linux operating system. In this operating system *first-touch* is the default data placement strategy to manage memory affinity on NUMA platforms. This policy places data on the memory bank of the NUMA node that first accesses it [Joseph 2006]. Due to this, it is important to guarantee that threads will touch their own data in order to enhance memory affinity. Therefore, data initialization in this case must be done before the iteration loop of our example to place data over the machine memory banks.

Figure 3.9 shows the difference between these two strategies in a NUMA machine with four nodes. One can observe, that with the master initialization all application data is placed in only one memory bank. Contrary to this, the team initialization data is spread over the four memory banks of the machine because each thread performs the first access on its chunk.

```
//Master Initialization
  for (j=0; j<N; j++)
  {
    c[j] = 0.0;
    a[j] = 1.0;
  }
```

```
//Team Initialization
 #pragma omp parallel for
      for (j=0; j<N; j++)
      {
        c[j] = 0.0;
        a[j] = 1.0;
      }
```

Figure 3.8: Data Initialization for the Synthetic Application.

Table 3.2 reports the execution time for each data initialization strategies. We compiled the code with GCC compiler and for the experiments we used four nodes of an AMD Opteron NUMA platform (similar to the machine in Figure 3.9). We used four threads and pinned each one to a core of the machine. As we can observe, execution times are up to 50% different, when using the master thread and the thread team initialization. In the latter case, data (array chunk) is placed closer to the thread that made the first access to it. This placement is performed by the *first-touch* strategy that used in Linux to place data on the physical memory of the machine. Considering the example, the team thread initialization reduces the number of remote access, minimizing latency costs to get data.

Table 3.2: Execution Time for the Synthetic Application (in seconds).

Data Placement	Execution Time
Master Thread Initialization	0.4992
Thread Team Initialization	0.2465

In this OpenMP case study, it is easy to identify and to reduce the NUMA impact because the initialization and the computation step have the same computation/data partition. Therefore, developers can apply the team thread initialization strategy in their applications in order to get better performances on NUMA machines. However, in real world applications computation/data partition generally change over the application phases. For these situations the worker thread initialization does not work. Thus, a mechanism to re-distribute data over the machine is necessary to take into account such an irregular accesses. To our knowledge, there is no mechanism with such feature in the OpenMP standard.

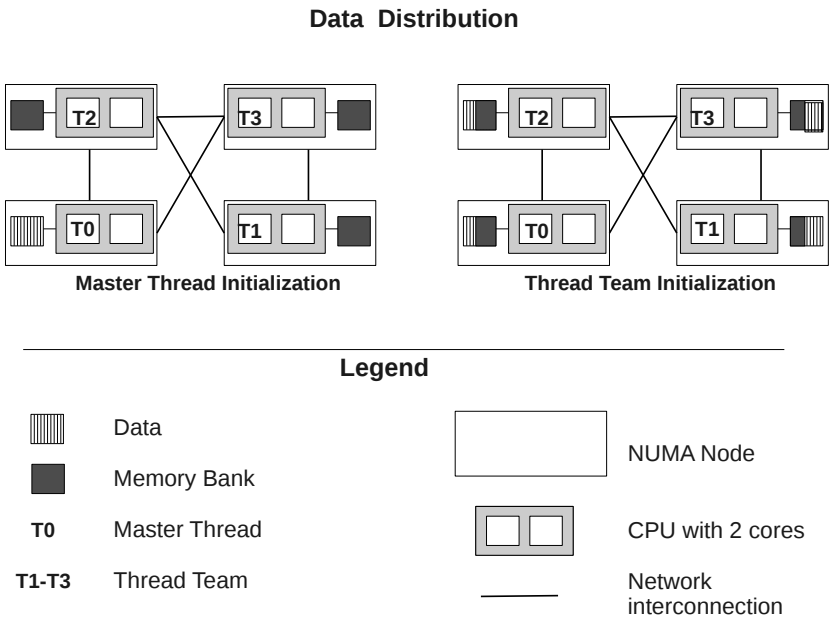


Figure 3.9: Data Initialization Strategies.

3.4 How to Reduce NUMA Impact on Parallel Applications?

In the above section, we noticed that a NUMA unaware interface/language such as OpenMP and Charm++ can have their performance reduced if developers do not take care of how thread and data are placed over the machine. A simple strategy already generates significant gains, as presented in the previous section with the team thread initialization. In this section, we introduce other problems related to the software level on NUMA machines. Additionally, we provide some hints of how users can reduce them by enhancing memory affinity.

The team thread data initialization strategy is generally efficient only in regular applications such as the example presented in Figure 3.7. In this case, memory access patterns are similar through the application steps, which allow developers to use this strategy effectively. However, in applications in which threads have a high level of data sharing and access different data, it becomes necessary the use of different strategies to place threads and data. To illustrate this problem, we present three case studies, the first one using a benchmark implemented with OpenSkel and the other ones using benchmarks in OpenMP.

An application that has a high level of data sharing between threads generally suffer from memory contention issues. Since threads try to access the same memory range at the same time, the available memory bandwidth for cores is generally not

enough. To illustrate this problem, we select the OpenSkel version of the *intruder* application from the STAMP benchmark [Minh 2008], which is an irregular application with a high level of data sharing between threads. We executed intruder with one thread per core on a NUMA machine with 32 cores and four NUMA nodes (8 cores per node). In order to reduce contention problems, we used an uniform round-robin data placement strategy that provides load balancing for data distribution, increasing the memory bandwidth for cores.

Table 3.3: Average Latency (ns) to get data on a NUMA Machine.

Number of Cores	Original	Round-Robin
8	233	220
16	2315	2286
32	5687	5586

Table 3.3 depicts the average latencies for a thread to get data. One can observe that by applying the round-robin data distribution, lower latencies are obtained when compared to the original version of the application. In this case, the application data is spread over all memory banks of the machine, increasing the memory bandwidth available for cores to access data. Since the threads have a high level of data sharing, the contention is reduced by providing more memory banks to be accessed.

A single application may need different data placements depending on the computation step and the memory access patterns of a variable. In Figure 3.10, we show a snippet of an OpenMP benchmark code, which illustrates different memory access patterns. The outermost loop (just after the #**pragma omp**) creates a set of OpenMP threads, each of which accesses the application matrices. In this case, each thread accesses a block of rows on matrix MD and all threads require access to common elements on matrix $\_negX$.

In order to avoid remote access on MD matrix rows, it is important to bind them close to threads that use them. To bind rows on the correct memory bank, we can use the team thread data initialization and let *first-touch* ensure that data is closer to threads that computes it. In this case, it is also necessary to pin threads to cores, to avoid any thread migration. The common elements on matrix $\_negX$ may be a source of contention if they are placed in a restrict set of memory banks, since all the threads are running simultaneously. That situation will happen if we use the team thread data initialization strategy to distribute data over the NUMA machine. To overcome the contention, one can then spread $\_negX$ in some round-robin strategy among the machine memory banks, in order to optimize interconnection network usage.

In Table 3.4, we present the execution time (in seconds) for the snippet shown in Figure 3.10 when executed in the four nodes of the NUMA machine described in the previous section. We use two strategies to place data for this code. In the first

```
#define N 13300

int MD[N][N],_negX[N][N];

#pragma omp parallel for private(j,radius,smallx)
for(i=0; i < _rows; i++)
 for(j=0; j < _columns; j++) {
    if (j == 0) MD[i][j] = 0;
    else {
           radius = 0; smallx = _negX[i][j - 1];
           while (radius < _gradius){
              if (_negX[i][j - radius - 1] < smallx)
                  smallx = _negX[i][j - radius - 1];
                  radius++;
           }
          if (smallx > _intX[i][j])
             MD[i][j] = 1;
          else
            MD[i][j] = 0;
    }
}
```

Figure 3.10: Synthetic Application Snippet.

strategy, we apply only the team thread initialization whereas in the second one, we combine this strategy with the round-robin one. The round-robin data placement strategy has been applied to $\_negX$ matrix by modifying the application source code.

Table 3.4: Execution Time in seconds for the Snippet Presented Above.

Data Placement	Time
Team Thread Initialization	4.19s
Team Thread Initialization + Round-Robin	3.39s

We can observe in this table that team thread data initialization performed 18% worse in time. Since this strategy considers only the first access to distribute data, it generates concurrent accesses to elements in the same memory bank. Contrary to this version, the team thread initialization + round-robin strategy perform an uniform round-robin placement for $\_negX$. Due to this, the memory pages are spread over the machine memory banks reducing memory contention on particular memory banks and increasing the available bandwidth. Considering the MD matrix, both versions of the code place its memory pages closer to threads that use them. Therefore, latency to get data is reduced by placing data on threads local memories.

Besides the memory access pattern (regular and irregular), it is also important to consider the data access mode (i.e. read and write) when choosing the data

placement strategy to enhance memory affinity. We have shown in chapter 2 that the write operations on NUMA platforms are generally much more expensive than the read ones. For each write on a shared variable, NUMA platforms have to ensure cache coherence for all cores of the machine, which increases the overhead for write operations. Variables accessed in read mode are generally replicated over the machine caches without additional costs to the coherence protocol. Therefore, data placement is also related to the mode that variables are accessed by threads.

In order to show the NUMA impact on data access mode, we performed some experiments with a tuned version of the Stream benchmark [McCalpin 2007], which is also implemented in OpenMP. In this tuned version, for each variable of the copy operation we applied a different data distribution considering its read/write access mode. In the case of read/write in the same variable, we changed the copy operation of Stream to use the same variable in the operation. The code of such benchmark is similar to the one presented in Figure 3.7. In this study, we have used a NUMA platform with eight nodes and sixteen cores. Furthermore, we used eight threads pinned to cores to better understand the NUMA impact considering latency (remote accesses) and bandwidth (network usage).

Table 3.5 depicts the execution time for the different versions of the Stream benchmark. We use three data placement strategies to reduce the NUMA impact: The *bind* one that places data closer to threads that use it; The *round-robin* one; And a third, that spreads data over the memory in a random way. We can observe that the three data placement strategies have reported different performances depending on the access mode. In general, the round-robin strategy is the best strategy for read operations. Considering the write operations, placing data closer to threads that access it is the best option for a static data distribution. These results let us to conclude that on NUMA machines the access mode require different data placement strategies in order to enhance memory affinity for a parallel region.

Table 3.5: Execution Time (us) for each Operation on Stream Benchmark.

Operation	Bind	Round-Robin	Random
Read	16	14	16
Write	207	269	310
Read/Write	225	285	334

Another point that must be considered in the memory affinity management is the work-sharing used to distribute work among the worker threads. Parallel languages and interfaces usually do it in a static or dynamic methods. In the former, each worker thread receives a similar amount of data to process. The work distribution is done statically for each parallel section. In the latter method, the amount of work is computed during the application execution considering threads availability. A number of studies have shown the importance of how work is distributed on NUMA machines [Bircsak 2000, Nikolopoulos 2001, Nikolopoulos 2002, Wang 2009]. These

studies show that different strategies may have a significant performance impact in the application.

3.5 Approaches to Improve Memory Affinity

In the previous sections, we noticed that modifications on the applications may be performed in order to enhance memory affinity and consequently, reduce load-balancing issues, memory contention and remote accesses on a NUMA platform. We also observed that in some cases, these modifications might be performed for each variable and parallel region, since they present different memory access patterns. Still, it is important to consider the characteristics of the NUMA platform (e.g. latency, bandwidth) in order to select the most suited memory affinity solution for an application. All these arguments must be considered for the design of efficient solutions to reduce the NUMA impact on application performance.

In this section, we present the state of the art on memory affinity management for NUMA machines. These solutions have been designed at different levels such as APIs, libraries, user tools, runtimes, compiler support and memory allocators. They can be grouped in: thread placement, data placement or a mix of both.

3.5.1 Thread Placement

Thread scheduling enhances memory affinity by mapping threads closer to their data and to other threads with which they communicate. In this type of solution, the placement can be performed statically, before the application execution or dynamically, during the application execution. In the dynamic placement, the scheduler can use runtime information regarding the platform and the application to place threads.

3.5.1.1 Preprocessing and Compile

The advent of multi-core machines with NUMA characteristics have brought new challenges to parallel compiler research groups such as threads scheduling. To deal with such a challenge, the compilers have been extended with thread affinity support. Several compilers have now some support to map threads over the machine cores, and consequently also improve memory affinity.

Considering OpenMP, there are some compilers that support thread placement. Examples of such compilers are Intel C Compiler (ICC) [1], GNU C Compiler (GCC) [2] and Portland Group Compiler (PGI). Using this support, developers can map OpenMP threads to the processing units of the NUMA machine. In this case, thread placement is done before the execution of the OpenMP application using environment variables or interfaces (few modifications on the source code are needed) that allow

1. ICC Thread affinity interface - http://software.intel.com/en-us/intel-compilers/
2. GCC Thread affinity interface - http://gcc.gnu.org/onlinedocs/libgomp/Environment-Variables.html

programmers to specify policies to distribute threads over the physical processing units.

Similar to the compiler support for OpenMP, thread affinity is also supported in TBB (Intel Threading Building Blocks [3]). TBB interface relies on the object oriented programming model. Due to this, the thread affinity support in this interface is based on a class that allows programmers to manage thread placement over the machine. The affinity for a team of threads is set by the user for a target application and machine. To do so, the user has to instantiate the basic class of TBB that provides the necessary mechanisms to set the affinity mask to be used for the team of threads.

In all solutions presented in this section, the affinity strategies are related to the machine topology and application. Therefore, the developers must be aware of these characteristics when choosing the affinity strategy for the parallel application. Since each machine has a different topology, it is mandatory for programmers to design a target thread affinity strategy for each machine. Therefore, the use of such interfaces restricts the application portability and requires from the programmers enough information about the machine hardware.

3.5.1.2 Runtime

In order to allow dynamic thread scheduling some research groups have integrated some support for thread placement into the language runtime. The authors of [Nikolopoulos 2001] presented a strategy to enhance memory affinity on OpenMP using the OpenMP clause *schedule*. In OpenMP, this clause is responsible for specifying how the work is distributed among the worker threads. The proposal is then to extract information of such clause to construct the relationships between a team of threads. For instance, how threads share data and how they access the shared memory. The work does not present any formal OpenMP extensions but shows some suggestions of how this can be done and what has to be included in the OpenMP API. The performance evaluation of the proposal was done using tightly-coupled NUMA machines. The results show that their proposal can scale well in the considered machines.

ForestGOMP is a runtime that proposes a dynamic thread placement for the OpenMP standard on NUMA machines [Broquedis 2009, Broquedis 2010a]. In this work, researchers have proposed an extension of GNU OpenMP library that relies on the *hwloc* framework [Broquedis 2010b], on the *Marcel* threading library [Namyst 1995, Danjean 2003] and on the *BubbleSched* framework [Thibault 2007]. This runtime uses hwloc to extract the target machine topology and then pin kernel threads on the machine cores. In order to provide more performance for OpenMP applications, the Marcel library is used to create user level threads within parallel sections and associates them to the kernel threads. Using the information provided by the user and the OpenMP constructions, the ForestGOMP runtime can better

3. Intel Threading Building Blocks Task-to-thread affinity - http://software.intel.com/en-us/blogs/2010/12/28/tbb-30-and-processor-affinity/

schedule the marcel threads over the NUMA machine. To do so, its scheduler uses the BubbleSched framework to provide it with the necessary structure to group threads and place them over the machine cores.

QuickThread is a library that allows development of parallel applications for multi-core machines using C++ and Fortran languages [Dempsey 2010]. It is a new programming paradigm that has constructions similar to OpenMP. Differently from OpenMP, QuickThread has explicit support for NUMA machines (e.g. data allocation functions, data placement). The QuickThread runtime implements efficient schedulers to ensure data locality at cache and NUMA node level. It also has an interface that allows programmers to define the affinity for threads, avoiding scheduling costs in cases where programmers have a good knowledge of the application and platform. Comparisons with OpenMP and TBB have shown that Quickthread can be profitable in NUMA machines [4].

The Charm++ parallel system also provides an explicit and implicit support to thread affinity. In the case of explicit support, a command line option allows programmers to set the thread placement for an application execution. The command line has to be used when launching the application, specifying which cores of the machine must be used to place the application threads. The implicit support is provided by the use of a load balancer that schedules work over the machine cores. The load balancer in Charm++ can be employed as a plug-in, using the load balancing Charm++ framework to build them. Due to the simplicity of design and implementation, a number of load balancers have been proposed for Charm++. For instance, load balancers that consider constraints such as memory usage and threads communication [Agarwal 2006, Koenig 2007, Bhatelé 2009, Dooley 2010].

The main limitation of these approaches is that they have been designed for a target language or interface. This prohibits their usage on other parallel languages that also fail in dealing with NUMA machines.

3.5.1.3 Machine Learning

The need for efficient and automatic mechanisms to map threads to the cores of a multi-core machine can not be put aside. While a number of works focus on the explicit control of the thread affinity (as presented above), only few studies have been done on automatic mechanisms to enhance thread affinity [Wang 2009]. Although the focus may be different, thread affinity mechanisms have always the same objective that is to improve performance by reducing the communication costs.

A machine learning approach to map threads over the machine cores is presented in [Wang 2009]. In this work, the authors have focused their proposal on parallel applications developed with OpenMP. Using the machine learning approach, the proposed solution is capable of automatically predict the number of threads and the thread placement policy for an application. The thread affinity mechanism has several steps in order to find the best number of threads and the thread placement policy for an application. First, the mechanism has to train the machine learning

4. http://www.quickthreadprogramming.com/

model with some target applications. For this step, the authors uses as input some characteristics of the application such as cache misses, loop iteration and branch miss rate. The output of the training step is a predictor for the best number of threads and scheduling strategy for a given application. After this step, the model is ready to be applied on applications not seen yet, using the application characteristics as input. The main limitation of this approach is that it can only predict correctly if the target applications have regular behaviors. If a novel application has to use this mechanism, it must have similar characteristics to the ones used in the training step. Otherwise, a new training must be performed to get the best thread placement for this application.

3.5.2 Data Placement

On NUMA machines, data placement strategies can also be used to improve memory affinity for applications. These strategies are usually implemented as specialized memory policies and NUMA-aware memory allocators. A memory policy defines how data is placed over the machine memory banks and the granularity used in this placement. It aims at improving data locality for a team of threads, but at the same time, it provides good bandwidth for data access. In contrast, NUMA-aware memory allocators use the machine topology to perform each malloc and free operation. This type of solution generally optimizes latency by allocating data closer to the thread.

3.5.2.1 Preprocessing and Compile

Data distribution and data placement strategies are also supported inside compilers of parallel languages and interfaces. Data distribution specifies which thread from a team of threads will work on a selected data. In contrast, the data placement describes how data is physically allocated on the NUMA nodes.

Both UPC and HPF have some directives that allow users to choose a data distribution for the worker threads of an application [Richardson 1996, Benkner 2002, Coarfa 2005]. Both languages have the block and cyclic strategies implemented as possible strategies to distribute the array elements among the team of threads. The *block* strategy splits an array in similar block sizes, considering rows and columns, and then attributes them to the worker threads. The *cyclic* strategy spreads the array among the worker threads in a round-robin way. However, on both languages, the data distribution strategies only deal with the logical elements of the arrays and not to their physical allocation on memory.

In [Bircsak 2000] the authors have presented new OpenMP directives that provide an efficient memory distribution for OpenMP. The new directives are based in HPF directives and they allow developers to express how data have to be allocated in the NUMA machine. The proposed directives are restricted to Fortran programming language. The authors present the ideas for the new directives and how they can be implemented in a real compiler.

On these proposals, the directives must be explicitly included in the application source code by the developer. This can be a difficult task, since the best data distribution is generally dependent on hardware characteristics. Therefore, developers must have prior knowledge of the platform to better choose the directive. Additionally, in the case of UPC and HPF, the selected data distribution is applied at the beginning of the application, without offering the possibility to change the physical data distribution over the application steps.

3.5.2.2 Hardware Guided

Most of current multi-core machines come with some support to extract hardware counters information. On NUMA machines, these counters (e.g. number of remote access, TLB misses) can be used to dynamically place data over the machine. Some solutions based in hardware counters have been proposed in [Tikir 2004, Marathe 2006, Awasthi 2010].

In [Tikir 2004], the authors introduced a profile-driven mechanism based on hardware counters to profile the memory access behavior of an application. This profile is then used to decide whether memory pages should be migrated. The objective is to increase the number of local memory accesses when possible. To do so, the proposed solution maps threads to processing units and then it profiles the application using a library. The mechanism uses helper threads to intercept the memory accesses in the profiling phase and then other group of helper threads are used to migrate memory pages to the selected memory banks. The performance evaluation using NAS Parallel Benchmarks presented a reduction of up to 18% of their total execution time and an overhead of up to 12% to profile and migrate application memory.

The work [Marathe 2006] presents a hardware-assisted page placement scheme based on automated profiling. The main objective of this solution is to reduce the execution time by placing memory pages closer to the most frequently requesting processor. The proposal relies on an automated profiling mechanism that extracts the application memory access patterns of both static and dynamic memory. Using such profiling information, some memory migration is performed to increase the number of local access. The proposed solution is implemented in the user space and it is independent of compiler, operating system and interconnection network but it relies on the machine providing the necessary hardware counters. This method has been evaluated using the NAS and SPEC OpenMP benchmarks and the results show that the mechanism can achieve performance improvements of up to 20%.

The recent integration of memory controllers inside processor chips has demanded a special attention when allocating data on machines based on such processors. Since memory controllers manage all access to physical memory, they can be a bottleneck and reduce the system performance. To deal with this problem, researchers have proposed in [Awasthi 2010] two memory policies that can adapt to improve data locality. The memory policies make use of the hardware information (e.g. queueing delay of a memory controller, number of hops from a core to

a memory controller) during the application execution to decide where to place a memory page. They have implemented the memory policies in the Virtutech Simics simulator [VIRTUTECH 2007] and used some benchmarks to evaluate them. Results have shown gains of up to 35% on platforms with one memory controller per N number of processors and gains up to 5% on platforms with one memory controller per processor.

The hardware guided solutions presented in this section are restricted to a set of platforms, reducing the solution portability and applicability. Furthermore, some of the solutions need some pre-execution to get information of the memory access patterns.

3.5.2.3 Memory Allocators

It is sometimes necessary to use dynamic data structures in the application. These data structures are usually allocated at runtime by the application using functions such as *malloc*. There are several different implementations of *malloc*, each one optimized considering one performance metric (e.g. allocation time and memory usage). For instance, *ptmalloc* [Gloger 2011], *tcmalloc* [Ghemawat 2011] and *hoard* [Berger 2000] are examples of *malloc* implementations.

NUMA-aware *tcmalloc* is a modification of the standard *tcmalloc* that provides scalable performances on NUMA machines [Kaminski 2009]. It is based on the NUMA API [Kleen 2005] and because of this, the NUMA-aware *tcmalloc* can only be used on systems that supports this API. The principle of the NUMA-aware *tcmalloc* is to assure that every *malloc*/*free* operation will be executed locally. This means that when a thread requires some memory, this memory allocator will consider the node where the thread is running to physically allocate data. In order to do so, NUMA-aware *tcmalloc* uses NUMA API functions to bind some memory range to the selected node. Experiments with synthetic *malloc* intensive benchmarks show that NUMA-aware *tcmalloc* improves in up to 37% the benchmarks performance. However, when it is used in real applications or more representative benchmarks (e.g. NAS Parallel Benchmarks) that are not *malloc* intensive, its performance is very similar to other memory allocators (e.g. original *tcmalloc* and *malloc* from *glibc*).

MAMI is a Marcel based memory interface that allows developers to manage memory allocation and placement for an application [Brice Goglin 2009]. MAMI provides some functions to perform memory allocation and deallocation that are very similar to the ones provided by any *malloc* implementation. All data allocation and deallocation in MAMI is NUMA-aware, since MAMI associates the requested memory to a NUMA node. Furthermore, it supports memory migration and memory statistics. In this proposal, memory management must be explicitly included in the application source code by the developer. This can be a difficult task, since the best data distribution is generally related to hardware and application characteristics.

3.5.2.4 Operating System Physical Data Allocation

NUMA support is now present in many operating systems, such as Linux, Windows and Solaris [Kleen 2005, Joseph 2006]. Since in this thesis we are mainly interested in NUMA machines with Linux as operating system, we focus our discussion on it.

On the Linux operating system, a basic support to manage affinity on NUMA is implemented as a kernel space data allocation. Physical data allocation is performed by using some memory policies to distribute data over physical memory banks. The standard memory policy used by Linux is *first-touch* but, other memory policies such as *on-next-touch* are now also available.

First-touch is the default policy in Linux operating system to manage memory affinity on NUMA. This policy places data on the node that first accesses it [Joseph 2006, Carissimi 2007]. To improve memory affinity using this policy, it is necessary to either execute a parallel initialization of all shared application data allocated by the master thread or allocate its data on each thread. However, this strategy will only present performance gains if it is applied on applications that have a regular data access pattern. In case of irregular applications, *first-touch* may result in a high number of remote accesses, since threads do not access the same data.

In the work [Löf 2005], the authors introduce a new memory policy named *on-next-touch* for Solaris operating system. This policy allows data migration when threads touch them on the next time. Thus, the threads can have their data in the same node, allowing more local accesses. Currently, there are some proposals of *on-next-touch* memory policy for Linux operating system. For instance, in [Terboven 2008, Brice Goglin 2009], the authors have designed and implemented the *on-next-touch* policy on such operating system. Its performance evaluation has shown significant performance improvements only when large blocks of data are migrated. This is due to the data migration overhead on Linux operating system.

3.5.2.5 Application Programming Interfaces

The NUMA support for the Linux operating system also provides a user space API to deal with data allocation and placement over the physical memory. The NUMA API is divided into two interfaces, a low-level one based on system calls and a high-level one based in a high level interface.

The kernel system calls defines some functions (*mbind*(), *migrate_pages*(), *set_mempolicy*() and *get_mempolicy*()) that allow the programmer to set a memory policy (*bind*, *interleave*, *preferred* or *default*) for a memory range (set of virtual memory pages). The memory policy specifies which physical memory banks will be used to place the memory range. A physical memory bank is composed of several frames, which are used to physically map the virtual memory pages. However, the memory policies only deals with the memory bank selection. The decision of which frame within a memory bank will be used to place the memory page is not made by the memory policy. The use of such system calls is generally a complex task, since

developers must deal with memory pages, sets of bytes and bit masks that identify memory pages and physical memory banks of the platform.

The second part of NUMA support in Linux is a library named *libnuma*, which is a wrapper layer over the kernel system calls. The set of memory policies provided by *libnuma* is the same as the one provided by the system calls. In this solution, the programmer must change the application code to apply the policies. The main advantage of this solution is that developers can have a better control of data allocation and distribution. However, similar to the system calls interface, *libnuma* demands source code changes to be employed. Additionally, no abstraction of the architecture is provided by the library. Developers must specify the memory banks that must be used for data placement.

3.5.2.6 User Space Tools

Some operating systems provide user space tools to deal with thread and data placement. Some examples are the *dplace* tool in Solaris, *numactl* in Linux and *hwloc* for several operating systems.

In the SGI NUMA machines additional support in the operating system for thread placement is available through the use of the *dplace* tool. This user space tool allows user to place threads over the machine processors/cores or nodes. In this case, *dplace* avoids any thread migration generally performed by the operating system, keeping the cache affinity.

The *numactl* tool allows the user to set a memory policy for an application without changing the source code. However, the selected policy will be applied over all of the application data. It is not possible to express different access patterns or change the policy during the application execution [Kleen 2005].

hwloc provides the user with an interface and a set of user space tools to deal with thread and data placement in a simpler way [Broquedis 2010b]. The interface has a set of functions that can be used to abstract the machine topology. Using this interface, programmers can retrieve the machine characteristics to then use *hwloc* tools to place threads and application data over the machine. However, this tool does not provide any support to extract the interconnection network information and NUMA penalties of a multi-core machine with NUMA design.

The main drawback of these tools is that they are platform-dependent. In order to improve memory affinity with such tools, the user must provide as arguments the list of nodes, the cores and the memory banks that will be used to place threads and data. Additionally, they do not allow the use of different memory policies within the same application execution.

3.5.3 Mixing Thread and Data Placement

The use of both thread and data placement mechanisms as solution for memory affinity relies on the idea of thread and data redistribution in the case of changes in the data access patterns. To our knowledge only one work has been proposed in

this context and it is described in [Broquedis 2010a]. In this work, authors combine two other solutions, ForestGOMP and MAMI [Brice Goglin 2009] to allow the use of thread scheduling and data placement to improve memory affinity. The approach demands that the developers provides some hints in order to correct any data placement. Developers have to change the application source code, to inform the runtime of ForestGOMP where the memory access patterns changes. In this way, ForestGOMP and MAMI can dynamically re-schedule threads and data placement in order to reduce latency to get data. This approach usually demands an integration in some language/interface runtime, which restricts its use to a target language/interface.

3.6 Conclusion

In order to ensure scalable performances on modern multi-core machines with NUMA design, the users must make use of all available support to efficiently manage the parallelism, thread scheduling and data placement. However, such management demands from users a good understanding of the machine hierarchy and the application memory access patterns.

Regarding the parallelism management, there are several mature languages and APIs that can be used to express the application parallelism (e.g. OpenMP, Charm). Several efforts have been done in the thread placement context. Current solutions inside operating systems and runtimes already present good performances (e.g. ForestGOMP). However, when it comes to physical data placement, we observed in this chapter that there is still lot of work to do. The state of the art solutions only deal with latencies issues on NUMA machines, avoiding completely the bandwidth management. Additionally, they do not provide an abstraction of the architecture to the application developer. As presented in the case studies, the NUMA characteristics of modern multi-core machines require from users a careful management of data placement that take into account both latency and bandwidths.

In the last two decades, a number of studies have been carried out in the context of memory affinity for NUMA machines, resulting in some efficient solutions. Most part of the proposed solutions have been designed for mono-core multiprocessors (e.g. HPF and UPC), which do not present the hardware issues introduced in chapter 2. Furthermore, most of these solutions rely on hardware counters and compiler support, resulting in a memory affinity solution dependent of the architecture/language. Therefore some questions remain:

- The multi-core machines are more complex and hierarchical than the mono-core multiprocessors. What is necessary to change in order to improve memory affinity for the multi-core machines?
- How can we abstract the multi-core machine with NUMA design architecture for the developers? How can we model these parallel machines?
- What level might be considered for an efficient memory affinity management, the application variables, parallel regions or the application heap?

– Should we consider the different memory access patterns of a parallel application? What are the impacts of this consideration?
– How do we match the architecture characteristics with the application characteristics, keeping the hardware independence?
– Can we avoid complex application source code modifications?

These questions are the main motivation of this thesis and answers for them are the objective of this work. Thus, we aim at providing efficient memory affinity mechanisms for modern multi-core machines with NUMA design. Our goal is to provide a memory affinity solution that is independent of the parallel language and machine.

Part II

Contributions: Looking Deeper to Improve Memory Affinity

Chapter 4

Proposal of New Approaches to Enhance Memory Affinity

In chapter 2, we have shown that modern shared memory machines are becoming more hierarchical and complex at the level of processing units and memory subsystem. This hierarchy brings new constraints to shared memory machines such as multiple cores sharing the machine resources, novel cache coherence protocols, complex interconnection network and hierarchical distributed shared memory. Due to these characteristics, memory access costs on these machines may vary depending on the distance between processing elements and memory banks, and based on the number of processing elements accessing the same memory bank. Therefore, to explore the potential and obtain good performances from these machines, it becomes necessary to have some software support to efficiently manage the usage of the memory subsystem of the machine.

High performance computing applications have different characteristics (e.g. memory access patterns), behaviors (e.g. regular, dynamic) and needs (e.g. latency or bandwidth). All of these characteristics must be considered when designing solutions to manage memory subsystem resources of modern shared memory multi-core platforms. In this context, enhancing memory affinity appears as a key element to match the application characteristics with the underlying architecture to improve the overall performance. Memory affinity is a relationship between threads and data that describes how threads access the application data. In particular, to enhance memory affinity it is necessary to keep data close to the threads which access it, to reduce the memory latency and memory contention perceived by threads.

As presented in chapter 3, several solutions have been proposed in the thread placement context for NUMA machines. However, considering physical data placement solutions some drawbacks still exist. Considering the current operating systems (e.g. Linux, Windows, Solaris), they do have some memory affinity support for NUMA machines. However, this support is general, since operating systems have to provide satisfactory performance for different types of applications (sequential and parallel ones). Because of this, they generally fail in providing an efficient memory affinity support for HPC applications. Still, in the last two decades, research groups have proposed specialized NUMA support inside languages/interfaces or machine hardware. Consequently, this support is usually dependent of the compiler, the machine architecture or both, which restricts their applicability. Furthermore, none of the state of art solutions contemplates the following properties: (i) different memory policies during the same application execution, not only one memory policy for

the whole application, (ii) system and machine portability, (iii) small or no code intrusiveness and most important (iv) different granularities for memory affinity management such as variables (e.g arrays and vectors) and application heap.

This chapter and the following two chapters introduce and describe the main contributions of this thesis. In this chapter, we introduce some ideas and concepts of how to enhance memory affinity for parallel applications by taking a deep look in the application and platform characteristics. Using these characteristics, we propose mechanisms to allocate and place data over the NUMA machines. Since memory affinity is a compromise between data and thread placement, we also consider thread placement in our mechanisms. For this, we use some classical state of art thread placement strategies such as compact and scatter to bind threads over the machine. After that, in chapter 5, we introduce our framework to control and manage memory affinity. Finally, in chapter 6, we show how we employ the components of this framework in parallel environments.

4.1 Modeling a NUMA Architecture

As depicted in chapter 2, the main characteristic of the current NUMA machines is their complex and hierarchical design. Processing elements, cache and memory subsystems of such machines are grouped in a hierarchical way, increasing the access latency and degrading the bandwidth usage. As a consequence to that, managing memory affinity (thread and data placement) efficiently becomes crucial to improve the performance of applications in such machines. Several characteristics of a NUMA machine, such as the number of nodes, the cache memory sizes, how cache memories are shared between cores and how the nodes are grouped, can be used as an architectural hint to enhance memory affinity. However, the choice of which architectural characteristics should be used on memory affinity management may be a difficult task and should not be relinquished solely to the programmers. Additionally, to attain portable performances on different NUMA machines it is important to propose memory affinity mechanisms that abstract the machine architecture.

In order to improve memory affinity on a NUMA platform with portable performances, the first step is to create a model that can represent the NUMA machine processing units topology and memory subsystem hierarchy. Therefore, we aim at designing a model that is capable of representing the memory access penalties present on NUMA platforms as well as the machine topology. In NUMA multi-core platforms, the NUMA penalty comes from different levels and due to this, we define two new terminologies, the NUMA core topology and the NUMA hierarchy. The first one is related to processing units, whereas the second one is related to the memory subsystem characteristics.

4.1.1 NUMA Core Topology

The NUMA core topology provides a view of how cores share resources and how they are assembled in the machine hardware. This information is important because

it expresses which processing units are sibling within a NUMA node, enabling a data and thread placement that is aware of the machine hardware. Therefore, we define as NUMA core topology, the physical organization of the NUMA nodes, the processing units and the cache memories inside the NUMA machine.

In the core topology a machine M is composed by a set of NUMA nodes, $N = \{N_0, N_1, ..., N_n\}$. Each NUMA node N_i is composed by a set of cores $C = \{c_0, c_1, ..., c_n\}$, that are organized in subsets. These subsets represent the cache memory sharing between the cores of the node N_i.

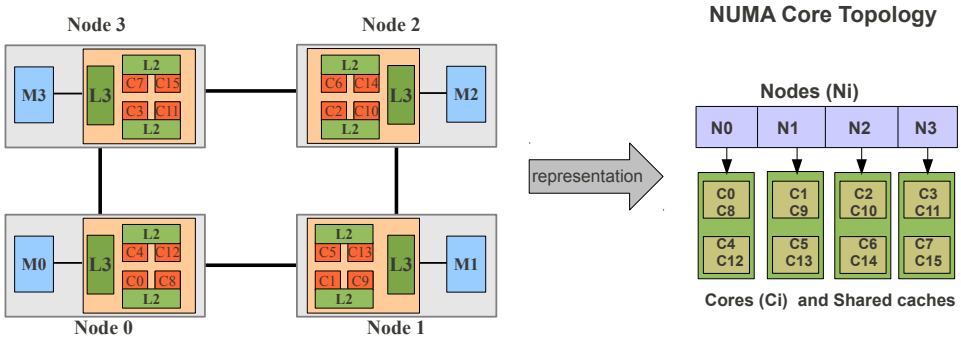


Figure 4.1: NUMA Core Topology for a NUMA Platform.

Figure 4.1 depicts our representation for the NUMA core topology of a multi-core machine with NUMA design. In the figure, our model shows that the machine is organized in four NUMA nodes and each one has four cores. Considering the cache memory hierarchy, the NUMA core topology model represents it by grouping cores that share some cache memory in the same subset. In the example, each two cores share a cache memory and each four cores share a last level cache. Therefore, for this example, each NUMA node has three subsets. A first one with the four cores that share the last level cache and other two ones, where each one is composed by the two cores that share the second level of cache. For instance, for the node N_0 of our example the three subsets are: $\{c0, c4, c8, c12\}$, $\{c0, c8\}$ and $\{c4, c12\}$.

4.1.2 NUMA Hierarchy

In contrast, the NUMA hierarchy represents the NUMA penalties related to the memory subsystem and the NUMA nodes physical organization. Representing the NUMA penalties related to memory accesses can be a difficult task, since it relies on different aspects of the machine (latency and bandwidth). In order to define a model that can represent this, we use performance metrics that allow us to express the asymmetry on memory accesses of NUMA machines. Furthermore, we select metrics that can be easily computed for different NUMA platforms using memory bound benchmarks.

The performance metrics used to represent the NUMA hierarchy are latency, NUMA factor and memory bandwidth. Considering the latency, we use the remote and local latencies of the NUMA nodes. These latencies allow us to identify which nodes are more distant, representing the machine interconnection network. Besides, these latencies are also used in our model to compute the NUMA factor.

We define the NUMA factor as the ratio between the remote and local latencies to get some data. The NUMA factor is a performance metric that allows users to have an idea of how costly a memory access can be in a NUMA machine. Therefore, the NUMA factor for the pair of nodes i and j is defined by:

$$NUMAfactor = \frac{ReadLatency \text{ from } \mathbf{i} \text{ to } \mathbf{j}}{ReadLatency \text{ on } \mathbf{i}} \tag{4.1}$$

The memory bandwidth is an important performance metric in NUMA machines because it specifies the machine memory throughput at some level. In our model, we decided to have the memory bandwidth for the intra-node and the inter-node links. This approach allows us to have a model that specifies the different bandwidths in the machine hierarchy. The intra-node bandwidth provides information for local memory bandwidth for a set of cores whereas the inter-node one represents the memory bandwidth between nodes.

Figure 4.2 shows our representation for the NUMA hierarchy, with the NUMA factor and bandwidth matrices. These matrices size is N by N, where N is the total number of NUMA nodes of the machine.

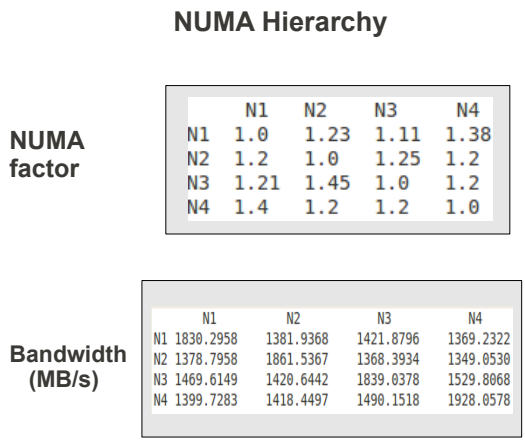


Figure 4.2: NUMA Hierarchy for a NUMA Platform.

Both NUMA core topology and NUMA hierarchy are retrieved by parsing information on the machine hardware characteristics and by running memory bound benchmarks. Implementation details for these mechanisms are provided later, in chapter 5.

4.2 Global Analysis of an Application

In this section, we take a look at the applications characteristics that can be used to better understand the application memory access behavior. Understanding the memory access patterns of an application let us better manage data placement and achieve higher performance over hierarchical multi-core machines. The investigation of the application memory access patterns is related to the application memory usage and not to thread placement.

4.2.1 What to Extract from the Application?

We are specially interested in studying numerical scientific multi-threaded benchmarks and applications, which exhibit significant memory and processing power usage, data-sharing between threads and various memory access patterns. The applications that we consider in this work follow the data parallel model with regular and irregular memory access patterns. In this parallel model, similar computation is performed over different data sets [Nyland 1996]. An application is regular if its memory access patterns are stable and predictable. Contrary to this behavior, in irregular applications, the memory access patterns are more complex and data dependencies are only know at runtime. For instance, OpenMP [OpenMP 2011], OpenSkel [Góes 2010a] and Charm++ [Kalé 1993] are languages extensions and interfaces that support the data parallel applications. In this thesis, we considered these languages extensions and interfaces as our case studies.

Generally, in these parallel languages, arrays are used as the main data structure for variables (e.g. OpenMP and Charm++). In this case, the parallelism is employed by splitting the array among a set of workers. Each worker then process its sub-array, synchronizing at the end to combine their results. Differently, the OpenSkel data parallel interface relies on dynamic data structures such as worklists. Although the data structure is different, the parallelism is also exploited by splitting the data structure among workers. More precisely, it assigns a set of elements (work-units) of the worklist to workers. The difference is that in this case the workload size and access are only know at runtime, making the application irregular. However, in all cases, workers have their own private and some shared data.

Applications developed with OpenMP and Charm++ usually implement several arrays within a single application. These arrays are used in different steps of the application computation and they are accessed in different ways. Since in each step of the application data accesses can be performed by a different worker and in different ways, various memory access patterns are expected for each array. Due to this, to enhance memory affinity for languages like that, it is important to consider each array separately in the memory affinity management. Otherwise, memory affinity is controlled for the whole application data and only a global view of memory accesses is considered. In the case of OpenSkel, the worklist has irregular memory access because each worker computes a set of work-units that encapsulates different dynamic data structures. Due to the irregular nature of dynamic structures, it

becomes crucial to manage memory affinity at the workers data granularity (work-units) for OpenSkel.

Our first step to enhance memory affinity is to identify the hotspot variables. They are usually the ones that demand memory and accesses the most. Generally, these variables are dynamically allocated, thus compilers can not do any optimization for their access and placement. Even if they are statically allocated, they may not fit in cache memories thus increasing the access latency to get them. These variables might even be so memory demanding that they may need more than one memory bank of the NUMA machine. Therefore, a careful data placement must be performed to avoid the use of an unbalanced set of memory banks, which can increase contention in the main memory interconnection network. Further, the most accessed variables are prone to contention themselves and are consequently more impacted by the NUMA machine design.

Although we believe that variables are the best units to deal with memory affinity, we also consider different granularities like the application heap and workers data for the memory affinity management. Such an approach allows us to better explore the application memory access patterns and the target machine resources.

4.2.2 Getting Memory Access Information from Applications

As presented in the previous section, for some of the considered languages the memory affinity management must be done at the application variables level. This strategy allows a better control of memory affinity because it deals with the various memory access patterns in a deeper way. We mean by deeper, a fine grain management of memory affinity that is applied for each variable of the application. In order to efficiently capture the memory access patterns of variables, it is important to define which characteristics from these variables must be considered.

We believe that the memory access characteristics that must be considered are the ones that suffer an impact from the NUMA design. For instance, the different costs of read and write operations on shared variables for NUMA machines due to the cache coherence protocol. In this context and considering the data structures used in our study cases, we select as characteristics the variable size, its access pattern (regular vs. irregular), its access mode (read only, write only and read/write) and its sharing mode (private vs. shared).

The variable size is taken into account because it influences the number of memory banks that are needed to allocate the application data. Additionally, this characteristic must be considered in order to ensure load balancing when using the machine memory banks and interconnection network. We also select the variable access pattern because it determines if variables should be split and the granularity used in this process. For instance, regular variables can be split in continuous chunks among the team of threads, since their accesses are performed sequentially by threads. As an example of that, we can cite the regular access pattern of OpenMP parallel loops with the static work-sharing. In this case, each worker thread computes continuous chucks of data to accomplish a task. Contrary to this regular pattern, irregular

variables experience complex and unpredictable memory accesses which increase the complexity for their distribution. In this case, smaller granularities such as memory pages should be used to avoid NUMA penalties.

The access mode is used because it is directly impacted by the cache coherence protocol and the interconnection network. Depending on the operation, different latencies and number of messages are generated on the machine by the cache coherence protocol. Finally, the sharing mode is also considered because it allows us to know which variables are shared or not. In the case of private variables, the placement mechanism can place them closer to the worker that owns them, since no other worker will need them. Differently, for shared variables their placement depends on the level of sharing among threads. In a low level of sharing, the team of threads only share few elements of the variable. Therefore, one can spread the variable on some memory banks, placing it closer to threads that access it more frequently. On the flip side with a high level of sharing, we can spread it on several memory banks to provide more bandwidth to get it, since migrating it on every different access may be expensive.

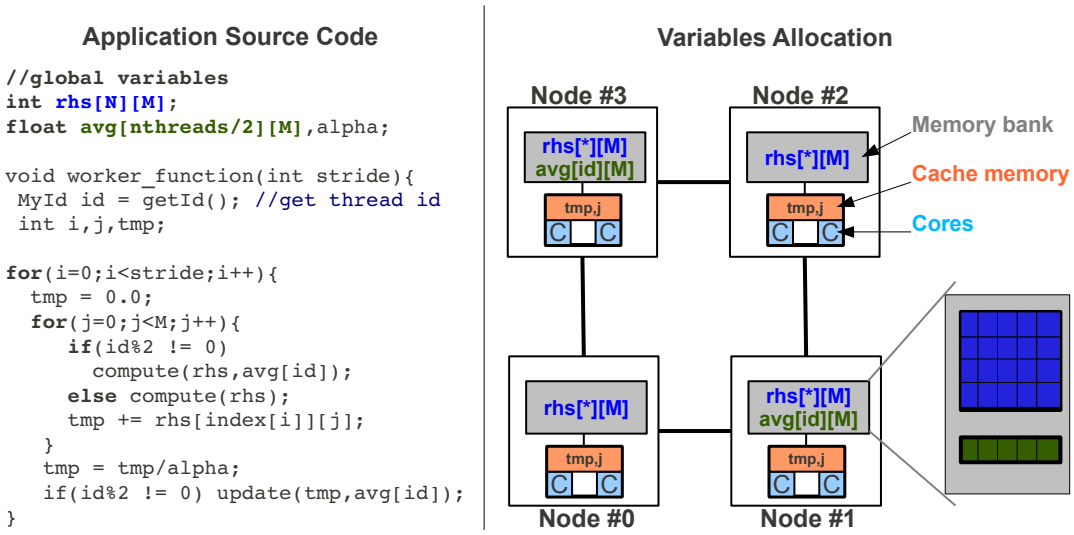


Figure 4.3: Variables Data Allocation over NUMA Machine Memories.

Figure 4.3 shows an example of a parallel application with private (i,j,tmp) and shared variables (rhs, avg and alpha). To enhance memory affinity for this code, it is necessary to identify which variables must be considered for the data placement. Following the characteristics described above, the variables that we have considered to memory affinity management are in blue and green. They are considered because of their size and sharing mode between threads. The Figure also depicts the mapping of the variables data in the NUMA machine memory subsystem. In this example, the NUMA machine is composed of four NUMA nodes. Each node is composed of a processor and a memory bank (gray rectangle in the figure). The processor has two cores (blue square) and a shared cache memory (orange rectangle). One can observe

that small data are placed in cache memories (*tmp* and *j*) whereas larger ones reside in main memory. Supposing that we run the application with four threads (one per core), the *avg* variable is split in one row per odd thread and this row is placed in the memory bank of the node where the thread is running. *rhs* is spread over all the machine memory banks because all threads access it.

4.2.3 Data Scope and Usage on Parallel Regions

In the memory affinity management, another important concept about the variable is its scope and usage within a parallel application. The variable scope defines its visibility in the different parts of the application, consequently defining its lifetime. The variable usage defines on which parts of the application the variable is accessed. Concerning parallel applications and NUMA machines, it is important to analyze scope and usage because these characteristics specify where an efficient memory affinity mechanism is necessary to in the parallel application to reduce the NUMA penalties.

In this work, we use the scope to determine whether a variable must be considered for memory affinity control or not. All variables that have global scope are evaluated if they should be managed. Generally in data parallel applications, these variables are shared among a team of threads and used within parallel regions to accomplish a job. Due to this, their influence in the application performance is significant. However, variables that are declared, allocated and freed within a function are not included in our memory affinity management, since their lifetime is restricted to the function scope. The exception are variables declared within the main function of an application. In this case, we consider these variables because in some applications shared variables are declared in that context. Examples of this situation can be found in data parallel applications developed with OpenMP, Charm++ and OpenSkel.

Particularly, the usage of a variable determines points in the parallel application where the memory affinity management must be employed. Since variables may have different memory access patterns in the parallel regions, it is important to know these points to avoid costly access latencies and bandwidths with suitable memory affinity control. Furthermore, the variable usage also allows us to identify where threads may need their data.

4.3 Associating Machine and Application Characteristics to Enhance Memory Affinity

In previous sections, we discuss what information is important to extract from a NUMA platform and from a parallel application. Now, we must work on this information to enhance memory affinity for parallel applications on NUMA machines. What should we do to achieve an efficient memory organization and data placement that considers both the machine topology and the application characteristics? This

section introduces one of our contributions: how to express data affinity, to organize and place data on an hierarchical shared memory multi-core platform. Since memory affinity is also related to thread placement, our contributions consider that threads are already bound over the machine cores, using some classical thread placement strategy. Therefore, any thread migration or scheduling is performed during the application execution.

4.3.1 Memory Organization: Why should we change it?

During the execution of an application, all of its data are placed into a virtual address space created for the process. This virtual address space is a memory management mechanism that allows the operating system to virtualize the physical memory for all applications running on the machine. When an application is started it receives a virtual address space that is divided into text segment, data segment, heap and stack. The text segment is used to store the application instructions whereas the data segment stores all data allocated at compile time. The heap and stack are used for dynamic memory allocation. The stack is used to store parameters and local variables of the application functions, whereas application data allocated with *malloc* are placed in the heap of the application address space. The mapping between the virtual address space and physical memory is latter performed by the operating system and the machine hardware using the strategy designed in the operating system. This strategy specifies which memory banks must be used to place a memory range.

We believe that in the case of languages with well defined data structures (e.g. arrays), it becomes crucial to have a specialized memory allocator implementation that organizes the application data on the heap, exploring the machine architecture. Therefore, considering the target languages and interfaces used in this thesis and the NUMA machine design, we propose two NUMA-aware memory allocators. The first one is dedicated to static data structures such as arrays, whereas the second one is designed to the dynamic data structures.

Static Data Structures: The static data structures are allocated in the data segment in the virtual address space of an application. The data segment is efficiently managed by the compiler that performs a number of memory optimizations in order to enhance data usage and access. However, the compiler sees the memory as a continuous space and it does not have any knowledge of the NUMA machine topology and hierarchy. In this context, compilers generally can not enhance memory affinity for such data. Furthermore, for arrays allocated in a dynamic way, using *malloc* functions for example, the compiler can not do any optimization because it does not know the data that will be allocated. Since arrays can be declared in static and dynamic fashion in parallel applications, it thus becomes necessary to ensure an efficient memory allocation for these data structures. To do so, we propose a specialized memory allocator for both static and dynamic arrays. Instead of using the data segment for static arrays, we allocate them in the heap segment that provides

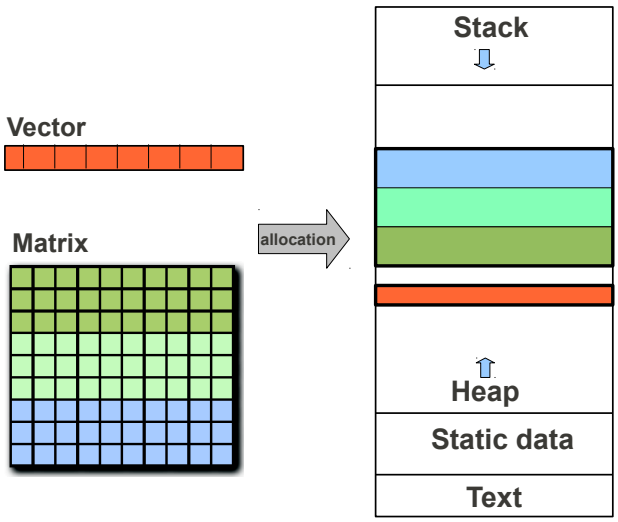


Figure 4.4: Array Allocator Design - Static Case.

a larger memory space to allocate such data. Additionally, heap segment enables memory affinity mechanisms to take the architecture into account at runtime, since they data of such segment is allocated during the application execution. Therefore, in our memory affinity mechanism all static arrays are automatically transformed in dynamic arrays. More details about this is presented in the next chapter.

Our array allocator relies on the principle that each array must have its own space in the heap. In this way, different arrays do not share the same blocks neither the same memory pages. This strategy reduces problems such as false sharing and reduces costs to deal with the memory affinity management for arrays. Since each array has its own separate space, elements of different arrays do not share the same block or memory page in the heap. It is also simpler to design a mechanism to deal with memory pages placement that has low overhead, since each array has its own space in the heap. Furthermore, a specialized array allocator allows us to simplify pointer manipulation for the programmer, reducing the burden of multidimensional array allocation. Figure 4.4 shows a schema of how our array allocator mechanism allocates memory for a vector and a two dimensional matrix. In this figure, the arrays are allocated in two separate blocks in the application heap, one for the vector and one for the matrix.

Dynamic Data Structures: Considering memory affinity management for dynamic memory allocation, it is interesting to deal with the application data that is allocated in the heap. This data usually consumes memory the most, and it can span a large lifetime, which demands a special control. Additionally, data allocated in the stack, which is also dynamic, generally has a small lifetime. Therefore, we consider only data allocated in the heap for memory affinity management of dynamic

data structures.

To allocate data in the heap, programmers have to use a memory allocator function such as *malloc* to request memory pages for storing it. Generally, in a memory allocator implementation, pages are requested from a pool of pre-allocated memory. This pool answers the request if it has enough memory and it places data in a heap block. However, if the pool does not have enough memory, the memory allocator requests more memory pages to the heap. In order to avoid too many requests to the heap and also to reduce memory consumption, the memory allocator also spreads new allocations among different heap blocks. In this way, different objects can reside in the same block and even in the same memory page [Berger 2000, Ghemawat 2011, Gloger 2011].

For the dynamic data structures (e.g. trees and linked queues), we then propose a different approach. Considering the dynamicity of these structures, the small granularity used in their allocations and the NUMA machine characteristics, we design a multi-level heap allocator. This memory allocator takes into account the characteristics of dynamic data structures and the NUMA machine hierarchy to optimize memory allocation, reducing memory consumption and time to perform allocations. In this case, the principle is opposite to the one used for the static data structures presented above. In this memory allocator, we let different objects share the same heap block and eventually the same memory page. Since an object size can vary significantly, we decided to optimize for memory consumption and memory fragmentation. The idea then is to use the concept of linked heaps (Figure 4.5(a)) already used in memory allocator implementations and create a multiple level design of linked heaps.

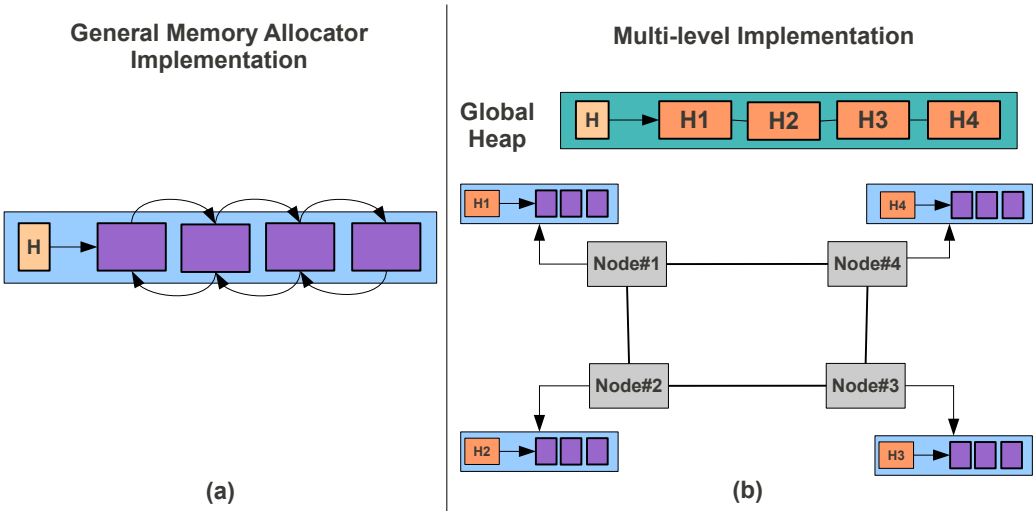


Figure 4.5: Memory Allocator Design - Dynamic Case.

In this context, each node of the NUMA machine in our mechanism has one heap associated to it and one global heap per machine is used to manage all other

heaps. The global heap is used to store information of the NUMA node heaps such as data distribution, free space and number of pages. In contrast, the NUMA node heap stores the application data. In this way, any memory allocation operations are made locally in the heaps associated to the NUMA nodes, reducing the NUMA penalties on the application.

Figure 4.5(b) depicts the schema of the multi-level memory allocator. In the example, we can observe a NUMA machine with four nodes. Each node has one heap associated with it that is used by all threads running in this node (represented in the figure by H1, H2, H3 and H4). The global heap is a centralized data structure that allow us to store and to retrieve information of all heaps of the machine (represented in the figure by H).

4.3.2 Memory Policies to Place Data

In order to enhance memory affinity using the information from the target NUMA machine and parallel application, it is necessary to define how application data allocated in the heap will be physically placed over the machine memory banks. A memory policy is used to specify how application data is distributed over the physical memory banks of the NUMA machine. In this work, the memory policy is responsible for unifying the architecture hints provided by our model with the selected application information to efficiently place data and, consequently, improve memory affinity. It is important to mention that in order to apply a memory policy to the application data, we consider that threads are already bound to the machine cores.

Considering the diversity of the NUMA machines design and the data parallel applications characteristics, only one memory policy might not be enough to enhance memory affinity. As presented in previous chapters and sections, different machines have different needs, as well as different applications have different memory access patterns. Therefore, in this thesis we exploit the usage of several memory policies into the same data parallel application on NUMA machines. It is important to mention that our memory policies assume that thread placement have been already performed using some classical strategy (e.g compact and scatter) and no thread migration is allowed.

Since the memory affinity problem on NUMA machines generally comes from the trade-off between latency and bandwidth issues, we propose memory policies that can handle both issues. Furthermore, in numerical scientific parallel applications, we cannot put aside the importance of some data structures, such as arrays, queues and trees. Most of the applications are represented using these data structures. Due to this, we also address in our memory policies different granularities for data placement in data parallel applications.

We then define two groups of memory policies, named bind and cyclic. The bind group aims at reducing access latency to get some data. It places data closer to the process/thread that uses it. The cyclic policies aim to balance memory banks usage and improve bandwidth to transfer data. It allows more memory banks to

be accessed in parallel, providing more bandwidth to cores. The bind group is composed of *bind all*, *bind block* and *next-touch* memory policies whereas the cyclic group is composed of *cyclic*, *cyclic block*, *cyclic neighbors*, *skew mapp* and *prime mapp*.

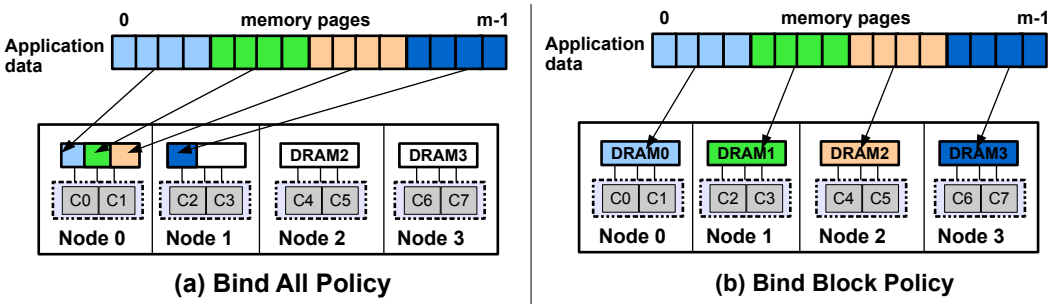


Figure 4.6: Bind Memory Policies.

Bind Memory Policies: In data parallel applications, the bind memory policies can lead to shorter access latencies to get data as long as threads and data are placed in the same NUMA node. Whenever a thread allocates a memory page, its corresponding virtual page is placed on a physical memory bank based on information about the NUMA topology and hierarchy.

Figure 4.6 depicts the *bind all* and the *bind block* memory policy in a NUMA machine with four nodes. Each node of the machine has two cores and one memory bank (DRAM). The application data is composed of m memory pages which are divided into four contiguous groups (each color represents a group). The *bind all* policy places all data in a selected set of memory bank(s). This policy will use all available memory (physical) from the first node memory bank before using the next one. The first node in this case is the node with more threads. In the case of *bind block*, each contiguous block is assigned to a different memory bank. The block size is defined considering the application characteristics. For instance, for an application that uses matrices, the block can be a set of rows or columns whereas for an application that uses dynamic data structures (e.g. trees) the block can be a memory range that represents an element of the data structure. The *bind block* policy can be profitable to data parallel applications in which each individual thread consumes its part of the shared data.

Finally, the *next-touch* policy places a memory page in the node that performs the next touch in it. In this case, the distribution is done considering threads accesses in the memory pages. While *bind all* and *bind block* are designed for data parallel applications based in arrays, *next-touch* is proposed for data parallel applications based in dynamic data structures (e.g. tree and queue).

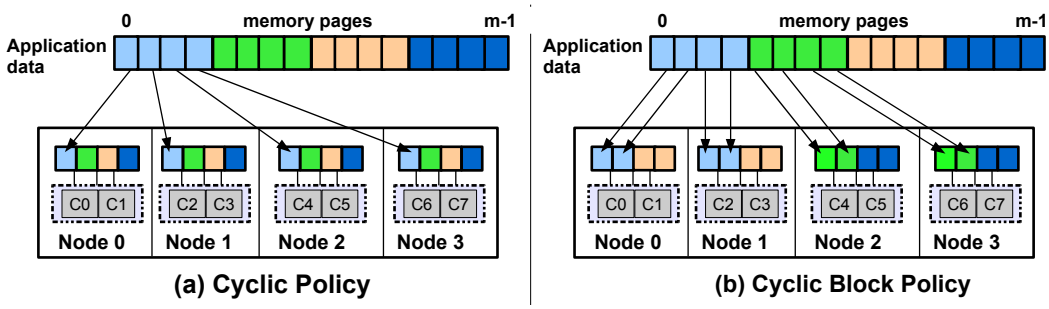


Figure 4.7: Cyclic Memory Policies.

Cyclic Memory Policies: A side-effect of the bind memory policies is that binding data to restricted memory banks may cause more memory contention when different threads share the same memory range. In order to avoid such behavior, the cyclic memory policies are employed. Cyclic policies spread memory pages over a number of memory banks of the machine following a type of round-robin distribution. In both *cyclic* and *cyclic block* policies, data is placed in the memory banks in a linear round-robin way. The first policy uses a memory page per round, a page i is placed in the memory bank i *mod* M, where M is the number of memory banks used by the application. In the second one, a block of pages b is placed in the memory bank b *mod* M. Figure 4.7 shows a schema that represents the *cyclic* and *cyclic block* memory policies in a NUMA machine with four nodes. Besides *cyclic* and *cyclic block* policies, the cyclic memory policies group is also composed of the *cyclic neighbors* memory policy. In this memory policy, the memory pages are also distributed in a linear round-robin way, using a memory page per round. However, differently to the *cyclic* and the *cyclic block*, this memory policy only places memory pages on the nodes that are neighbors of the node that request data placement. Neighbors nodes are the ones that have direct connection to the node of the caller thread. The *cyclic neighbors* memory policy also optimizes bandwidth, but it also keeps data locality since data is placed nearby the caller thread.

Cyclic, *cyclic block* and *cyclic neighbors* memory policies can be used in applications with regular and irregular behavior that have a high level of sharing, since it provides more bandwidth and better memory banks usage. However, some scientific applications can still have contention problems with these memory policies, because these strategies make a linear power of two distribution of memory pages on a platform that also has power of 2 memory banks. Since data structure sizes used in scientific numerical applications are also power of two, the *cyclic* distribution may place memory pages that are used by different threads in the same memory bank [Iyer 1998].

To overcome this problem authors have proposed in [Iyer 1998] two non-linear round-robin strategies, the *skew mapp* and *prime mapp* memory policies. These memory policies aim at reducing concurrent access on same memory banks for

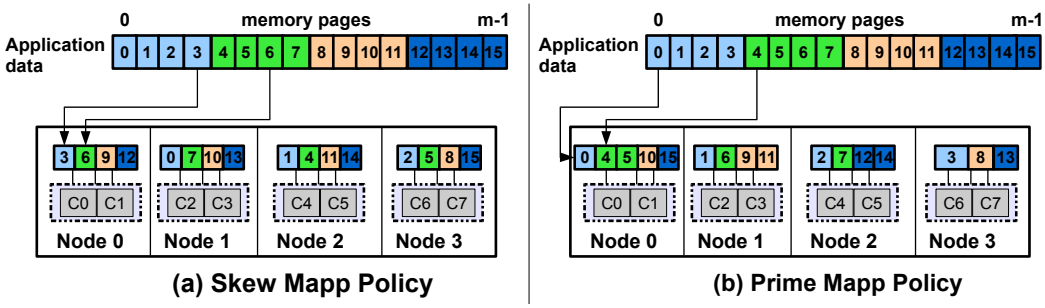


Figure 4.8: Cyclic Memory Policies.

parallel applications, by performing non-linear page placement over the machines memory banks. The *skew mapp* memory policy is a modification of round-robin policy that has linear page skew. In this policy, a page i is allocated in the node $(i + \lfloor i/M \rfloor + 1)$ *mod* M, where M is the number of memory banks. In this case, the *skew mapp* memory policy is able to skew a node in every round of the data distribution. Therefore, memory pages are distributed over the machine memory banks in a non-uniform way. Figure 4.8 (a) shows a schema that represents the *skew mapp* policy in a NUMA machine with four nodes.

The *prime mapp* policy uses a two-phase round-robin strategy to better distribute memory pages over the NUMA machine. In the first phase, the policy places data using *cyclic* policy in (P) virtual memory banks, where P is a prime greater than or equal to M (real number of memory banks). Then, in the second phase, the memory pages previously placed in the virtual memory banks are re-placed into the real memory blocks also using the *cyclic* policy. In this way, the real memory banks are not used in a uniform way to place memory pages. Figure 4.8 (b) depicts a schema that represents the *prime mapp* policy in a NUMA machine with four nodes and five virtual memory banks.

All the memory policies presented in this section supports data migration. This means that different memory policies can be applied to the same data (variable or memory range) on different phases of the parallel application. In this case, the selected memory policy migrates the memory pages over the machine memory banks in order to ensure the policy strategy for the selected data.

As aforementioned in chapter 3, some languages and interfaces such as UPC, HPF and OpenMP have already employed some of the memory policies introduced in this section. For instance, UPC, HPF and OpenMP extensions implement *bind block* and *cyclic* data distribution. However, in the case of UPC and HPF, they do not perform physical data placement over the machine memory banks like our memory policies. Both, UPC and HPF only distributes the logical elements of an array among the worker threads. Additionally, to our knowledge none of the state of art works on OpenMP have employed *cyclic block*, *cyclic neighbors*, *skew mapp* and *prime mapp* memory policies to place application data on the physical memory

banks of the NUMA machine.

Besides, these solutions only apply memory policies to arrays, while our memory policies can be applied to arrays, to a memory range or to the application heap. A number of memory policies support are attached to a single parallel language/interface, which allows their usage in only a specific set of applications. Additionally, they do not allow the use of different memory policies in the same set of variables through the application steps. Once a memory policy is applied for a variable it remains until the end of the execution of the application. Our proposal provides the possibility to use several memory policies in the same variable (data migration), applying different memory policies for the variable depending on the application step. In this case, the memory policy applied to a variable is changed to a new one.

4.4 Data Placement over NUMA Machines

As described in the previous sections, in order to efficiently manage memory affinity for data parallel applications on NUMA machines, information from both architecture and application must be considered. Using such information, the application source code can be explicitly or implicitly modified to enhance memory affinity. In this work we provide a flexible memory affinity solution that supports both explicit or implicit mechanisms.

4.4.1 Explicit Approach

Our explicit approach relies on the programmer knowledge of the application to manage memory affinity on parallel applications. Using the proposed memory allocators and the memory policies the programmer can manually change the application source code to apply a strategy that is most suitable for the application and the architecture.

In order to do these modifications, the first step is to select the appropriate memory allocator for the application. Parallel applications based on arrays should use our array memory allocator whereas the other ones should use our general purpose memory allocator (the multi-level heap allocator). After that, for each variable of the application, the programmer must verify its memory access pattern, access mode and sharing mode, in order to select the variables that must be included in the memory affinity management. For each selected variable, the programmer has to change the application source code to include its dynamic allocation and its memory policy or policies. The chosen memory policies must match the variable memory access patterns with the machine characteristics. By doing this for the application steps, the user have in the end a NUMA-aware parallel application.

4.4.2 Automatic Approach

Expert programmers who have a deep knowledge of the machine characteristics and also of the applications needs can manage memory affinity in an explicit way,

using our memory allocators and memory policies. However, this may demand some modifications in the application source code. Therefore, in order to provide a more transparent memory affinity management, an automatic support may be employed for parallel languages.

We propose an automatic solution to manage memory affinity that uses the NUMA core topology - NUMA hierarchy defined in section 4.1, the application characteristics depicted in section 4.2 and the memory policies described in the above section. It aims at removing from the programmer the responsibility of managing memory affinity in parallel applications by automatically modifying the application source code.

However, the problem of placing data over the machine memory banks is similar to the scheduling problem, which is known to be NP-complete [Leung 2004]. Several parameters from the application and the NUMA machine must be considered to place data over the machine. Therefore, it is not possible to compute a data placement on memory banks that optimally enhances the memory affinity in polynomial time (unless P = NP).

In order to cope with the complexity of the data placement, we have developed a new heuristic that matches the NUMA architecture characteristics with the application memory access characteristics to produce a NUMA-aware application source code. We chose to use an heuristic algorithm because heuristics provide good approximations in a reasonable amount of time. The heuristic algorithm is based in some empirical studies presented in [Ribeiro 2008, Ribeiro 2010a].

The heuristic selects the application variables that must be considered for the memory affinity management using the application information described in section 4.2. Its input arguments are the list of the variables information (named Var), the NUMA hierarchy (named $NUMAh$) and the NUMA topology of the machine (named $NUMAt$) presented in section 4.1. For each variable in Var, the heuristic has the variable size, its sharing mode (private vs. shared), its access mode (read vs. write) and its access way (regular vs. irregular). In the $NUMAh$ input, we represent the machine hierarchy using the NUMA factor whereas, the $NUMAt$ provides information of the machine hardware.

Figure 4.9 shows the heuristic algorithm to choose the most effective data placement for a variable considering its memory access characteristics and the underlying NUMA characteristics. First, the heuristic loads the variables of an application to the Var list and the machine information to $NUMAh$ and $NUMAt$ (lines 2-4). Then the heuristic selects from Var only the eligible variables, which are the large and not private ones (lines 5-9). We mean by large variables, the ones that do not fit in the second level cache of the architecture (line 6). We do not consider small variables because compilers generally do several optimization on them and they generally fit in cache memories. Private variables are not considered because generally compilers create a temporary variable for each process/thread that accesses them.

A new list named Var' is then generated. Next, the heuristic decides if the variable has to be bound closer to the thread that access it or if it has to be spread over the machine memory banks. In order to do so, the heuristic verifies the average

```
ALGORITHM: Heuristic

1 INITIALIZATION
2    Add all variables into Var
3    NUMAh <= load NUMA_factor
4    NUMAt <= load NUMA_Topology

5 FOR ALL Var(i) IN Var
6     IF Var(i) > NUMAt.cache_size AND
7       Var(i).share_mode diff PRIVATE
8          Add Var(i) into Var'
9 END FOR

10 WHILE Var' is not empty
11   Remove Var'(i) from Var'
12   IF NUMAh < THRESHOLD
13    IF Var'(i).access_mode equal WO AND
14     Var'(i).access_way equal REGULAR
15       bind(Var'(i))
16    ELSE
17       cyclic(Var'(i))
18   IF NUMAh >= THRESHOLD
19    IF Var'(i).access_mode equal RO AND
20     Var'(i).access_way equal IRREGULAR
21       cyclic(Var'(i))
22     ELSE
23       bind(Var'(i))
24 END WHILE
```

Figure 4.9: Heuristic Algorithm to Automate Data Placement.

NUMA factor of the machine (line 12) and the variable access mode (line 13) and access pattern (line 14). High NUMA factors implies potentially longer access latencies whereas low ones implies shorter access latencies. Therefore, for low NUMA factors the variables are generally spread over the machine to enhance memory bandwidth, except for write-only variables accessed in regular pattern. Write-only variables generally generate longer latencies, due to the cache coherence protocol of the machine. Therefor, even on a NUMA machine with low NUMA factor these latencies can have significant impact on the application performance. For high NUMA factors the variables are generally bound over the machine to reduce access latency, except for read-only variables accessed in irregular pattern. Since the memory access patterns of read-only variables accessed in an irregular way are not predictable, spread these variables over the machines provides better bandwidths to get them. Additionally, place these variables closer to threads demands data migration, which in this case will happen several times.

4.5 Summary

The NUMA design inside the modern multi-core machines has been implemented as a scalable solution to cluster their memory subsystems. Multi-core machines with

NUMA design keep the abstraction of a single shared memory by implementing on-chip memory controllers and cache coherence protocols. However, this solution generally brings costly access latency and low memory bandwidth to get data. Furthermore, NUMA machines present a complex core topology and memory subsystem hierarchy.

In order to extract scalable performance from these machines, memory affinity thus becomes a key element. Particularly, memory affinity keeps data closer to threads reducing the impact of the NUMA design in the parallel applications. Since parallel architectures and applications have different characteristics and needs, it becomes necessary to enhance memory affinity for these applications by applying efficient and suitable data placement strategies.

We believe that an efficient memory affinity mechanism has to consider the multi-core machine and the application characteristics. However, it must not be dependent of the application language/interface and machine architecture hardware characteristics such as hardware counters. The dependency restricts the use of the mechanism on few target parallel languages and machines. Besides, extracting, selecting and exporting such characteristics should not be relied only to programmers, since it can be a laborious task. Therefore, from our point of view, a memory affinity solution for multi-core NUMA machines must be architecture and compiler independent, support explicit and automatic mechanisms.

The main contributions of this thesis are: (i) to provide a portable way to model the NUMA platform, to represent its topology and hierarchy; (ii) to organize and to select the most important application memory access pattern characteristics; (iii) to improve memory affinity with memory policies applied to the different application data and steps; (iv) to design and to implement a framework to manage memory affinity for parallel applications with portable performances.

In the next chapter, we introduce and describe Minas Framework, which is a portable and efficient solution to manage memory affinity on data parallel application over multi-core NUMA machines.

Chapter 5

Minas: a Memory Affinity Management Framework

In this chapter, we introduce Minas, a memory affinity management framework for cache-coherent NUMA multi-core platforms that implements the concepts presented in chapter 4. Minas provides an explicit memory affinity management mechanism and an automatic one that are independent of machine architecture and compiler [Ribeiro 2010b, Ribeiro 2010d]. The explicit tuning is based on a portable API named MAi (Memory Affinity interface) which provides simple functions that allow programmers to manually manage data allocation and placement using an extensive set of memory policies. An automatic tuning mechanism is provided by the preprocessor named MApp (Memory Affinity preprocessor), which analyses the application source code in order to automatically apply MAi functions in the source code. Both MAi and MApp use information of the target cache-coherent NUMA platform to better manage memory affinity. Such information is provided by a Minas component named numArch. We start the chapter presenting an overview of the Minas framework. After that, for each Minas component, we present its design and implementation details.

5.1 A Framework to Manage Memory Affinity

Minas is an efficient and portable framework that allows developers to manage memory affinity on parallel applications for large scale NUMA platforms. We mean by efficiency a fine control of memory accesses for application data and similar performance on different NUMA platforms. As portability, we mean architecture and compiler abstraction and none or minimal application source code modifications.

5.1.1 Software Architecture

Minas is composed of three components: MAi, MApp and numArch. MAi, which is a high level interface, is responsible for implementing the explicit NUMA-aware application tuning mechanism whereas the MApp preprocessor implements an automatic NUMA-aware application tuning. The last module, numArch, extracts several information about the target platform, which is then used by the MAi and the MApp components.

Figure 5.1 shows a schema of the Minas approaches to enhance memory affinity. The original application source code can be modified by either using the explicit

mechanism (red arrows) or the automatic one (black arrows). In the case of the explicit mechanism the programmer has to change the application source code to manually improve memory affinity. In contrast to this approach, in the automatic mechanism the application source code is automatically changed by Minas. The decision between automatic and explicit mechanisms depends on the developer's knowledge about the target application and platform. One possible approach is to first use the MApp automatic tuning mechanism and to check whether the performance improvement is considered sufficient or not. If the gain is not sufficient, the developers can then explicitly modify (manual tuning) the application source code using MAi. Both MAi and MApp rely on numArch to retrieve some of the machine hardware information and its memory subsystem performance.

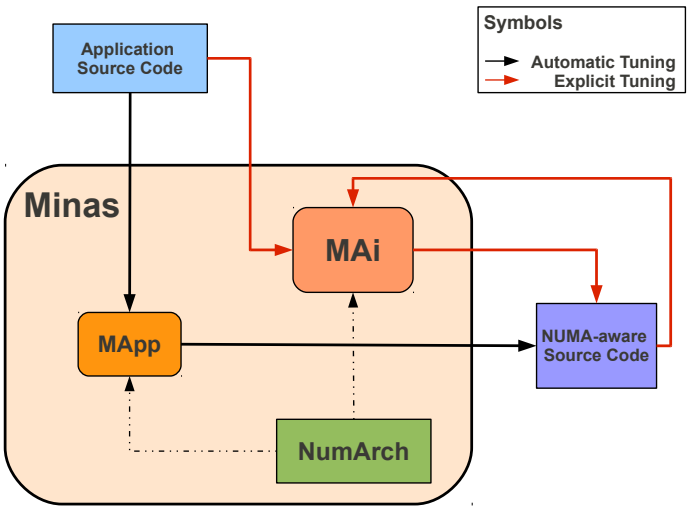


Figure 5.1: Overview of Minas.

5.1.2 Components

In this section, we describe the main characteristics of each Minas component and its importance in the framework. Their implementation details are later described in section 5.2.

The **numArch** module has an important role on Minas, since it retrieves the machine information that are necessary to place data on memory banks and to pin threads on cores. This component extracts information about the interconnection network (bandwidth), memory access costs (e.g. NUMA factor and latency) and architecture characteristics (e.g. number of nodes, cpus/cores and cache subsystem). It can also be used as a library, since it provides some high level interface that can be used by the developer to better understand the machine topology and its memory hierarchy characteristics [Ribeiro 2010b, Pilla 2011a]. In Figure 5.2, we present the numArch components and the output description file for a NUMA machine.

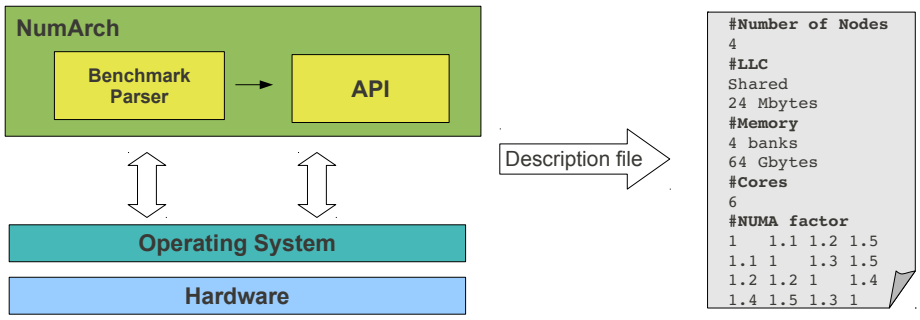


Figure 5.2: Overview of numArch Module.

MApp (Memory Affinity preprocessor) is a preprocessor that provides a transparent control of memory affinity for numerical scientific parallel applications written in C. MApp performs optimizations in the application source code considering the application memory access patterns and platform characteristics at compile time. Its main characteristics are its simplicity of use and its platform and compiler independence. This preprocessor has been developed in cooperation with the University Federal of Rio Grande do Sul [Ribeiro 2010c, Ribeiro 2010e].

Figure 5.3 shows a schema of the MApp process to enhance memory affinity for applications. The process starts with information extraction of the application variables. In turn, the original application source code is parsed by the MApp parser, named CUIA (Code Under examInation to retrieve informAtion) [Stangherlini 2010]. After that, it fetches the platform characteristics, retrieving information from the numArch module. Then, the MApp heuristic matches this information with NUMA hardware characteristics to choose the memory policy for each variable. Afterwards, the MApp code transformation module modifies the application source code.

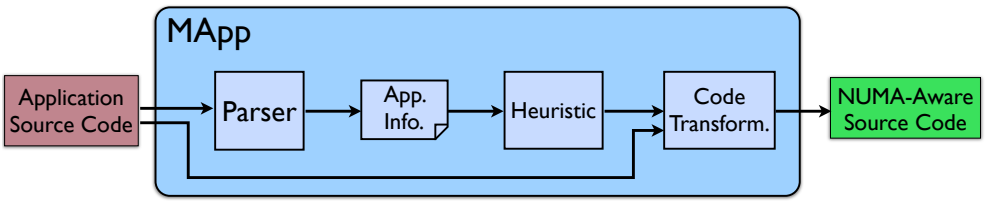


Figure 5.3: Overview of MApp code transformation process.

MAi (Memory Affinity interface) is an API that provides simple mechanisms to deal with memory affinity on NUMA platforms. It provides simple and high level functions that can be called in the application source code to perform data allocation, placement and migration [Ribeiro 2009a, Ribeiro 2010d, Góes 2011]. MAi functions can be divided in three groups: allocation, memory policies and system functions.

Allocation functions are responsible for reserving space for application data on

heap, similar to a standard malloc function. Memory policy functions are used to physically distribute data among the memory banks of the machine. MAi implements the memory policies introduced in the previous chapter that can be used to optimize memory access on NUMA platforms (latency and bandwidth optimization). System functions allow developers to collect and print system information such as memory banks used by the memory policies, cpus/cores used during the application execution and statistics about data migration.

Regarding thread placement, MAi implements some classical thread placement mechanisms that are used in some scenarios to better manage memory affinity. Furthermore, MAi can use application memory access traces to pin threads on the machine cores. In the current version of MAi, memory affinity can be managed at two different levels: an array based and a more general purpose one. The array based one must be used for applications based on arrays and parallel loops whereas, the general one is used for parallel applications that rely on dynamic data structures. The choice of which API from MAi should be used relies on the application developer. Figure 5.4 shows some of MAi functions for its two designs.

```
/*Initialize MAi*/
int mai_init(char *file);

/*Allocate arrays*/
void* mai_alloc_1D(int nx,size_t size);
void* mai_alloc_2D(int nx,int ny,size_t size);
void* mai_alloc_3D(int nx,int ny,int nz,
                   size_t size);

/*Free arrays*/
mai_free(void *array);

/*Memory policies*/
int mai_cyclic_neighbors(void *array);
int mai_prime_mapp(void *array);
int mai_bind_rows(void *array);

/*Finalize MAi*/
int mai_final();
```

(a)

```
/*Initialize MAi*/
int mai_init(char *file);

/*Allocate data*/
void* mai_malloc(size_t size);
void* mai_malloc_local(size_t size);

/*Free data*/
mai_free(void *ptr);

/*Memory policies*/
int mai_cyclic_neighbors(void *ptr);
int mai_next_touch(void *ptr);
int mai_bind_local(void *ptr);

/*Finalize MAi*/
int mai_final();
```

(b)

Figure 5.4: Some Functions of MAi Interfaces: (a) array functions (b) general functions.

5.2 Implementation Details

In the previous section of this chapter, we described the main functionalities and components of Minas. In this section, we present some implementation details of Minas components.

5.2.1 Extracting Platform Information

On hierarchical shared memory machines with NUMA characteristics there are two types of information that are important to retrieve in order to have a good

knowledge of the platform and provide architecture abstraction to Minas. The first one is the machine topology, which describes how processing elements share memories and how they are related to each other. The second one is how the non-uniform memory access impacts on the platform performance.

The machine topology consists of a set of information that describes its hardware. Considering Minas objectives and scope the following information is necessary: number of NUMA nodes, number of cpus/cores, number of sockets, number of caches, size of cache memories, set of cores that share a cache memory, memory banks size, free memory and relation between nodes and cpus/cores. To retrieve such information, numArch parses the topology-related /sys/devices/ and /proc/PID/ file system of the Linux operating system. From the file system, numArch gets the information of the hardware of the machine parsing some text files. From these text files , numArch extracts the information of the nodes (number and physical id), of the cores (number, physical id and siblings cores) and of the cache memory hierarchy (number of levels, size and sharing among cores). For instance, in a Linux operating the folder /sys/devices/system/node/ provides information of all NUMA nodes of the machine whereas, the text file /sys/devices/system/cpu/online allows us to know the number of cores online in the machine. Figure 5.5 shows the folders and files that numArch uses to create the machine topology description.

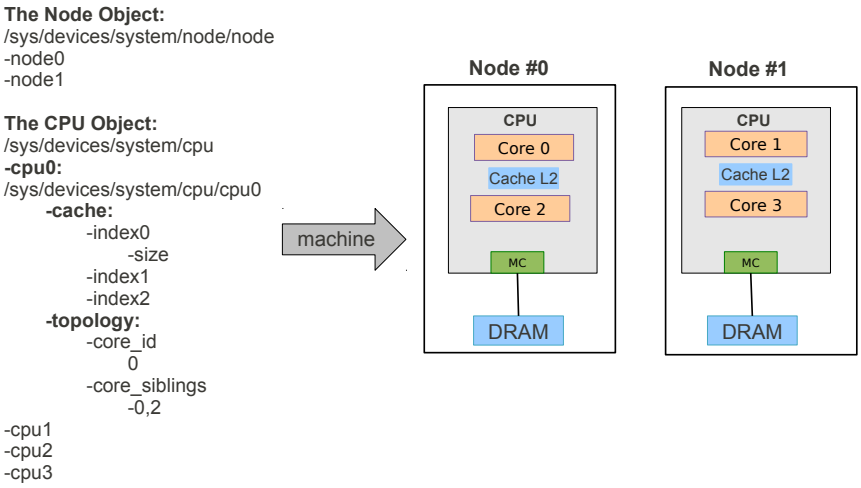


Figure 5.5: Operating System File System Information used by NumArch.

After parsing step, the obtained information is stored in temporary files on the /tmp/ of the machine. In the initialization of Minas, these files on /tmp/ are loaded to dynamic structures (e.g. queues, hash tables and matrices) that can be later accessed using the numArch interface (Figure 5.6). For instance, nodes ids and cores ids of the machine are organized in a hash table indexed by the node id. For each entry of the hash table, there is one node id i and some core ids, the ones that physically belongs to the node i. It is important to mention that is probably necessary to re-implement some of these functions because of the differences between

operating systems and architectures.

Since Minas deals with data placement over the machine memory banks, it is also important to obtain information of NUMA hierachy between nodes of the platform. In order to do so, we use two well know benchmarks, Stream [McCalpin 2007] and LMbench [LMbench 2010]. Stream is a synthetic benchmark application that measures the aggregate memory bandwidth for a machine. LMbench is a set of synthetic benchmarks that measures scalability of multiprocessor platforms and the characteristics of the processor micro-architecture. From LMbench, we selected the benchmark *lat_ mem_ rd*, which allow us to compute the read latency to access data allocated in a node.

```
/*is it a numa machine*/
int na_is_numa();

/*get number of nodes of the machine*/
int na_get_maxnodes();

/*get number of cpus/cores per node*/
int na_get_cpusnode();

/*get the NUMA node id which a cpu/core
belongs to*/
int na_get_nodeidcpu(int cpu);

/*get the amount of free memory per node*/
unsigned long na_get_memnode(int nodeid);

/*Get cores that share a cache level with
core*/
int* na_get_cacheShare_cores(int core);

/*get bandwidth between two nodes*/
double na_get_bandwidth(int nodei,int nodej);

/*get latency between two nodes*/
double na_get_latency(int nodei,int nodej);

/*get numa factor between two nodes*/
double na_get_numafactor(int nodei,
                         int nodej);
```

Figure 5.6: Some Functions of NumArch Interface.

During the installation of Minas in the machine, numArch run an application that performs benchmarking of the machine by executing the Stream and LMbench between each pair of nodes. After that, numArch saves the bandwidths and latencies between nodes in temporary files. NumArch also computes the NUMA factor using latencies obtained from LMbench for each pair of nodes, which is also stored in text files in /tmp/ directory of the machine. These memory performance metrics can be retrieved on any application using the functions *na_ get_ bandwidth(int nodei,int nodej)*, *na_ get_ latency(int nodei,int nodej)* and *na_ get_ numafactor(int nodei,int nodej)* from the numArch interface.

5.2.2 Extracting Application Information

In the Minas framework, we have chosen to implement our preprocessor to achieve compiler independence. MApp retrives application information using a two level parser. The first one extracts information of variables whereas the second one extracts information of the parallel constructions of the programming interface.

The parser responsible to extract information of variables is written with Lex/Yacc tools and it is called CUIA. It aims at providing variables information of the parallel application. CUIA parses C code and returns, for each of the variables: (i) its name and type; (ii) its lexical scope, and the name of the file where it has been declared; (iii) its *nature*, which can be either "static array" (*e.g.* `int x[10];`), "dynamic array" (*e.g.* `int* x;`), or "scalar". When the variable is a static array, CUIA also obtains the number of dimensions of the array and the number of entries in each one of the dimensions; (iv) a list of the access modes made on the variable: Read, Write, or both, and (v) the location in the program where these accesses occur (line/column number of the `for` instruction in which scope the access is performed). Regarding access modes, a variable is considered as read when it appears as right value, and as write when it shows up on the left of a equal symbol. No inter-procedural analysis is performed by CUIA. It restricts its analysis on each parallel region of the application.

This information is retrieved from different parts of the code being parsed. Some are trivial to obtain (*e.g.* the name and type), other require some more semantic actions in the Yacc grammar. CUIA uses the ANSI C Yacc grammar, published in 1985 by Jeff Lee [1]. Only the rules that are relevant to obtain the required information have been completed with semantic actions. A symbol table has been implemented to manage this information, which is classically initialized when the declarations are parsed, and afterwards complemented with the extra information, for instance the access modes. At the end of the parsing, the content of the symbol table is dumped in a textual format, compatible with the later phases of MApp.

The Figure 5.7 presents the output of CUIA for an entry consisting in the LU factorization source code (Fig. 5.7(a)). For each variable, one can see in Fig. 5.7(b) the nature, the type, the number and sizes of the dimensions of the arrays, the scope and filename where it has been declared. For the variables that are arrays (Minv, I and M), the list of access modes is also returned, with the information on the `for` construct which uses them (line and column in the source code file).

The second level of the parser is implemented inside MApp and it retrieves information of how data is distributed for worker threads (work sharing) and the sharing type of a variable (i.e. shared vs private). The input for this parser is a text file with the list of variables generated by CUIA. For each variable in the file the parser verifies its sharing type and the work sharing for the parallel region the variable belongs to. The sharing type and work sharing are obtained from the parallel language/interface constructions. For instance, in the OpenMP example presented in Figure 5.7, the standard defines that any variable used in the parallel

1. See for instance http://www.lysator.liu.se/c/ANSI-C-grammar-y.html.

```
1  #define N 1024
2  int M[N][N], I[N],Minv[N][N];
3
4  void main(){
5    int rows,diagonal,sum,col,tmp,pivot;
6
7   for(pivot=0; pivot < N-1; pivot++){
8     #pragma omp parallel for private(tmp,col)
9     for(rows=pivot+1; rows < N; rows++){
10        M[rows][pivot] = M[rows][pivot]/
11                         M[pivot][pivot];
12        tmp = M[rows][pivot];
13        for(col=pivot+1; col < N; col++)
14           M[rows][col] = M[rows][col] -
15                         M[rows][col]*tmp;}}
16
17   #pragma omp parallel for private(diagonal,sum,I,col)
18   for(rows=0; rows < N; rows++){
19    for(diagonal=0; diagonal < N; diagonal++) {
20     sum = 0;
21     for(col=0; col < N; col++)
22       sum = sum + M[diagonal][col]*I[col];
23     I[diagonal] = I[diagonal] - sum;}
24     for(diagonal=N-1; diagonal >=0; diagonal--) {
25       sum = 0;
26       for(col=N-1; col >=0 ; col--)
27         sum = sum + M[diagonal][col]*M[rows][col];
28       Minv[rows][diagonal] = (I[diagonal] - sum)/
29                              M[diagonal][diagonal];
30    }}
31  }
```

(a)

```
'rows' NOT_ARRAY 'int' 'main' 'lu.c' -
'diagonal' NOT_ARRAY 'int' 'main' 'lu.c' -
'sum' NOT_ARRAY 'int' 'main' 'lu.c' -
'col' NOT_ARRAY 'int' 'main' 'lu.c' -
'tmp' NOT_ARRAY 'int' 'main' 'lu.c' -
'pivot' NOT_ARRAY 'int' 'main' 'lu.c' -
'Minv' ARRAY_DIM 'int' 2 1024 1024 'global' 'lu.c'
      (W,24,3,'lu.c')
'main' NOT_ARRAY 'void' 'global' 'lu.c' -
'I' ARRAY_DIM 'int' 1 1024 'global' 'lu.c'
      (R,22,3,'lu.c')
      (RW,23,3,'lu.c')
      (R,28,5,'lu.c')
'M' ARRAY_DIM 'int' 2 1024 1024 'global' 'lu.c'
      (RW,10,6,'lu.c')
      (RW,14,9,'lu.c')
      (R,22,3,'lu.c')
      (R,27,4,'lu.c')
      (R,28,3,'lu.c')
```

(b)

Figure 5.7: (a) Input C code. (b) CUIA output.

loops is shared. The private ones must be explicitly declared in the *private* clause of the OpenMP directive. Regarding to the work sharing, in OpenMP the default is the static one (each thread receive similar chunks of data). A dynamic work sharing is obtained in OpenMP using the clause *dynamic*. The work sharing information is saved in a structure that is associated with the variable. This structure is later used to decide what memory policy to use for a variable in a parallel section. The sharing type is used to select or not the variable to be controlled by Minas. If the type is private, the parser removes the variable from the text file (in the example the vector I is removed). On the contrary the variable will remain in the text file and later modified by MApp in the application source code.

5.2.3 Allocating Memory for Applications

In Minas, data allocation relies on the MAi library memory allocator implementation. As mentioned before, we have two levels of memory allocation inside MAi. The first one that allows user to allocate arrays (MAi-array) and a second one (MAi-heap) that supports any type of object allocation. During the desgin of MAi, we decided to separate it into two levels because we wanted to do some optimization for arrays structures. A number of parallel languages, libraries and applications (e.g. Charm++ [Kalé 2009b] and OpenMP [OpenMP 2011]) are based on this type of data structure. However, some support for generic data allocation is also needed on High Performance Computing systems (e.g. Software Transactional Memory and OpenSkel [Góes 2010b]).

To implement these memory allocators, we defined two separate interfaces. However, both implementations relies on *mmap()* system call. This system call allows us to reserve some allocate a separate non-contiguous of virtual memory for a process and its threads. We decided to use this system call because it allows us to reserve a separate area for the new memory allocation in the application heap. Additionally, other functions that also reserve memory for applications such as *sbrk()* are becoming obsolete, since it demands from the memory allocator more control to index the allocated area. Since the parallel applications used in this work demand a large amount of memory, it becomes necessary the use of such system call to reserve memory on the application heap. Additionally, in our array based memory allocator implementation, we want to provide separate areas for each array.

5.2.3.1 MAi-Array

In the array memory allocator, each time that a new array is requested with one of the functions *mai_ alloc_ *D(int dimensions,size_ t size)* one call to *mmap()* is performed, allocating $dimensions \times size$ of memory. After that a pointer to the array is returned to the application. MAi-array supports up to four dimensional arrays, but it can easily be extended to support more dimensions. The dimensional number of elements is a mandatory parameter of MAi-array memory allocator. Due to this, the user must specify the number of rows, columns and plans for each array.

The memory allocation in MAi-array is always aligned with the memory page size of the system to simplify the data distribution over the memory banks by our memory policies. For each allocation, a header that contains information about the array is also allocated and it is stored in MAi-array control structures (see Figure 5.8). This information is later used at runtime by MAi-array to retrieve metrics about performed allocations and memory usage for an application.

The pointer returned by the *mai_ alloc_ *D(int dimensions,size_ t size)* functions can be directly used as an array, because MAi-array organizes all levels of pointers and elements of the array. For instance, an allocation of a two dimensional array, *int **a=mai_ alloc_ 2D(int nx,int ny,size_ t size)*, provides to the user a pointer to the array that can be accessed directly by its indexes (e.g. a[0][1]). For each allocated array, we store its information in a hash table addressed by some bits of the array address. The hash table provides an efficient mechanism to search arrays information when we have to apply a new memory policy to the array or free it (*mai_free(void *array)*), for instance. Any collision in this hash table is treated with linked lists.

In order to provide an idea of memory allocation costs considering time to allocate data and total memory consumption of the application, we present bellow results for five arrays sizes with MAi-array. The synthetic benchmark used in this experiment is single threaded and allocates two arrays with two dimensions. Time to allocate includes time to reserve memory and to set all pointers for the two dimensional array. The sizes used in this experiment are the same ones used in the applications we later use to evaluate the performance of Minas. Results are com-

```
/*Information about variables*/
struct var_info{
  int mem_policy;
  int ndim;
  int nnodes;
  int dimensions[MAXDIM];
  size_t size;
  //mutex lock to guarantee exclusive access to the variable header
  //by the threads that uses the variable
  pthread_mutex_t lock_var;
  //nodes in which the variable pages are physically allocated
  int *nodes;
  void *phigh; //pointer to the array
  void *plow;  //pointer to the array elements
};

/* MAi malloc statistics */
struct mai_malloc {
        size_t used_size;
        size_t nmmaps;
        size_t block_h;
        int npinfo;
};
typedef struct mai_malloc mai_stats;
```

Figure 5.8: MAi-array Header

pared to the standard memory allocator of glibc in a Debian distribution of Linux operating system (version 2.6.32-5-amd64).

Table 5.1: Time in microseconds and Virtual Memory Consumption

Size	MAi-array		Linux	
	Time	Memory	Time	Memory
64 x 64	8	23156 Kbytes	3	12280 Kbytes
256x256	8	24140 Kbytes	8	13204 Kbytes
1024x1024	17	27444 kbytes	23	16376 kbytes
4096x4096	34	278 Mbytes	64	268 Mbytes

We can observe in Table 5.1 that considering time to allocate memory, MAi-array present better performance than the glibc allocator for larger array sizes. The design of MAi-array memory allocator leads it to simplify the memory allocation operation, since it does not have to perform search operations for free space in the heap. Looking at memory consumption, MAi-array memory allocator has performed worse than the glibc implementation. Differently from glibc, MAi-array does not try to find free spaces in the previous allocations. It always allocates a new aligned

space in the heap for the new array. Due to this, MAi-array should not be used for the allocation of very small arrays. However, when we increase the size for allocation our malloc implementation memory consumption approaches that of glibc. Since MAi-array is proposed for data parallel applications and NUMA machines without intensive malloc allocations, this limitation is not important.

5.2.3.2 MAi-Heap

In the MAi-heap interface, the hierarchical memory allocator mechanism (introduced in chapter 4) is implemented using different levels of heaps. For a NUMA platform, MAi-heap defines a global heap that encapsulates all the memory allocation information of the machine and also one heap per NUMA node. Therefore, in our implementation for each node a *mmap()* call is performed to create a pre-allocated area in the application heap for the node. In order to associate this new area to the node, we use the system call *mbind()*, which allows us to specify where the physical memory pages of the new area must be placed. In this case, we perform n calls to the *mbind()* function, where n is the number of NUMA nodes of the machine.

This design provides more parallelism to the MAi-heap, since threads do not have to wait for a centralized heap to respond its memory space request. Figure 5.9 shows a schema that represents this implementation. In the figure, the global heap has four heaps, one per NUMA node, that points to a node heap. Each node heap has several sub-heaps that holds the application data. This sub-heaps are created considering the application memory allocation requests. Every time that an allocation can not be satisfied to a NUMA node a new heap is created and attached to this NUMA node. Heaps within a node heap are connected using double linked list to enhance performance of *malloc* and *free* operations, reducing time to search for a memory range.

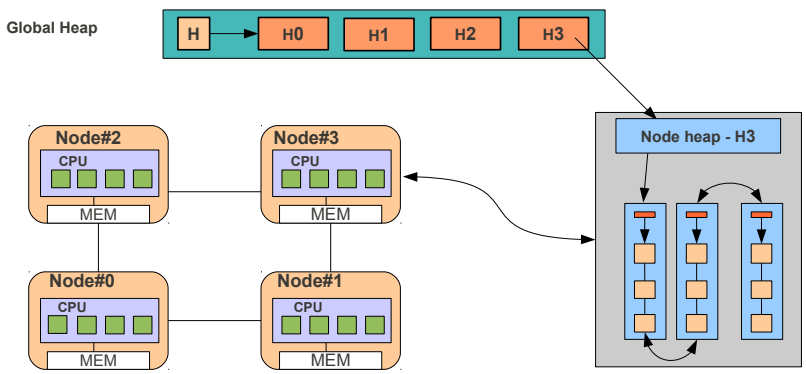


Figure 5.9: MAi-heap Implementation View.

The heap associated with the node is the one that really contains data that was allocated by threads. The global heap in the Fig. 5.9 is used only to store information to control data allocation. For each allocation request with the function *mai_ malloc(size_ t size)*, the global heap select which node heap should respond. To do so, the global heap verifies where the caller thread is running (with *sched_ getaffinity()* or *pthread_ setaffinity()*) and designates a node heap to respond that allocation. If the node heap has enough memory to satisfy the requisition, it increases its used size, updates node and global heaps and returns to the caller thread a valid pointer. On the contrary, a new node heap is allocated and linked with the existed ones. This new heap will respond the caller thread allocation request.

The MAi-heap is also aligned with the memory page size of the system, but at the level of the node heap. Each time a new heap is created it is aligned and it has a minimum number of memory pages that are requested (depends on the target platform). This strategy reduces overhead with system calls, because it minimizes the number of necessary calls to the *mmap()* function. In order to update the node and global heap, we have a mutex lock for each heap level to reduce the granularity of each critical section. The global heap has one mutex lock, which is only used if any information of the node heaps must be updated. The node heap mutex is used to avoid concurrent accesses in the heap, when several threads request data.

The performance of the MAi-heap for each memory allocation, considering costs in terms of time to allocate is evaluated with a synthetic benchmark. In this example, a multithreaded application allocates inside each thread two lists of N integers. For this experiment, we have used one thread per core on a NUMA machine with 32 cores and four nodes. Results are compared to the standard memory allocator of glibc in a Debian distribution of Linux operating system (kernel version 2.6.32-5-amd64).

Table 5.2: Time in microseconds to allocate Two Lists of N Integers

N	MAi-heap	Linux
10	9	11
100	11	11
1000	30	50
10000	42	97

We present time to perform the memory allocation for the MAi-heap and the Linux standard memory allocator (Table 5.2). Considering the time to perform an allocation, we can observe that Minas has presented better performances that the Linux standard memory allocator. Since Minas memory allocator reserves some initial space on the heap during its initialization, the initial mallocs are very fast because they are satisfied by this pre-allocated area.

5.2.4 Placing Data over NUMA nodes

Data placement on Minas framework is implemented as memory policies inside MAi-array and MAi-heap. Due to this, the object used on every data placement can be an array or a memory range of a heap. On both cases, the granularity used by MAi to place data over the memory banks is always a memory page. As presented in chapter 4, there are two groups of memory policies, the ones that optimizes latency (*mai_bind_*()* and *mai_next_touch()*) and the ones that optimizes bandwidth (*mai_cyclic_*()* and *mai_*_mapp()*).

In order to implement the latency memory policies on both MAi interfaces, we divide a memory range (an array or a heap) into N chunks (e.g. rows, columns, memory pages) and place these chunks on the memory banks where threads are running. Linux operating system provides a set of system calls that allow programmers to bind virtual memory pages on physical memory banks. These system calls receives as parameters the pointer to the virtual area, the number of pages to be placed on the memory bank and the physical id of the NUMA node that must be considered in the bind operation. There are two types of system calls, the ones used to bind a memory range to a memory bank (*mbind()*) and the ones used to migrate a memory range already bound to a memory bank (*move_pages()* and *migrate_pages()*). Therefore, to place these chunks on the physical memory, we use the system call *mbind()* or *move_pages()*.

The *mbind()* system call allows us to specify where a memory range should be placed on the NUMA platform. However, if that memory range has been already placed on any memory bank, MAi uses *move_pages()* system call that migrates the memory range to the destination node. The *mbind()* system call also supports data migration but, because of implementation issues, its performance is generally worse than the one obtained with the *move_pages()* function. Because of this, in MAi memory policies implementation every memory range already mapped to physical memory are always placed using *move_pages()* function.

The bandwidth memory policies are also implemented using the *mbind()* and *move_pages()* system calls. The difference here is that those memory policies use a set of nodes for a memory range placement and not only one NUMA node. Furthermore, the memory range is generally divided in a set of memory pages (considering cache memory hierarchy) instead of chunks. The choice of which set of nodes to use for a memory range depends on which thread asked for the data placement and the memory policy distribution. Considering these information and the topology of the machine (e.g. NUMA nodes), MAi looks for nodes that can receive the memory range. For instance, in the *mai_cyclic_neighbors()* policy, MAi retrieves the NUMA node where the thread is running and then using the machine topology (NUMA factor and physical NUMA node id), it extracts the neighbor nodes to place data.

Figure 5.10 shows a code snippet of the *mai_cyclic* memory policy. In the figure, *do_bind()* function is responsible for calling the *mbind()* system call and ensure that only one thread will perform the operation. To do so, it first acquires the lock for the

```
void mai_cyclic(void *p)
{
  void *realptr;
  unsigned long *node;
  int i;

//retrieve the low pointer:
//p is the high level pointer to the array, but
//the memory policy needs the address of the memory range
  realptr = get_realpointer(p);

//in cyclic all memory banks of the machine are used
  set_all_nodes(node);

//if nodes have enough memory
   if(free_memory(node) > realptr->mem_size) {
//for each page of the memory range
    for(i=0;i<realptr->npages;i++)
      //place a page i in the node i % number of nodes
      do_bind(realptr->padeaddr[i],node[i]%get_max_nodes());
   }
   else{
    printf("\n Node does not have free memory.");
    return (-1);
   }
}
```

Figure 5.10: Cyclic Memory Policy Code Snipet.

correspondent address and then, it places the memory page in the target memory bank. Since this policy always uses all nodes available on the machine, it does not have to check for which thread asked for data placement.

On every data placement, MAi always checks if the destination node has enough physical memory consulting numArch module. If the destination node has enough memory, data is placed on the node. On the contrary, MAi searches for neighbors nodes that can receive the memory range. In order to get the neighbors of a specified node, MAi asks to numArch using the function *na_get_neighbors()*. This function uses the NUMA factors computed at Minas installation to define which nodes are closer to the one that does not have enough memory. Data placement is finalized with an update to MAi data structures, in which the nodes used to place the memory range are saved inside these structures. This information is important because MAi has to decide between *mbind()* or *move_pages()* to physically allocate data.

5.2.4.1 How to choose a Memory Policy?

As mentioned before, there are two possibilities to choose a memory policy for a memory range using Minas, an explicit one and an implicit one. In the explicit method the developer changes the application source code by himself using MAi-

array or MAi-heap interfaces in order to provide hints to Minas of which memory policies to use and where to apply them. To do so, programmers only have to use the functions of MAi interface presented in the previous sections.

In the implicit approach, Minas relies in the MApp heuristic (chapter 4) to automatically modify the application source code. This heuristic is implemented as a module of MApp and it uses as input the CUIA output file and the numArch machine description file. Since CUIA provides as output a list of all variables of the application, the heuristic has to select which variables it will manage the memory affinity. It only control variables that present an important impact on memory affinity of an application, i.e. variables with a long lifetime and size at compile time.

In C, variables can be allocated in three ways: statically, automatically and dynamically (malloc). In this work, we are mainly interested in static and dynamic variables, since automatic ones are temporary objects stored in the stack that will be used in a restricted scope. Static variables have the same lifetime as the binary of the application whereas dynamic variables are alive until they are freed. Thus, on NUMAs, these variables will be more impacted by the data distribution. A variable is considered large if its size is equal or greater than the size of the second level cache of the platform (extracted from numArch module). This choice was taken because small variables are generally kept in caches. So, their impact on memory affinity strategies are not significant. Furthermore, the MAi functions have some costs related to data allocation and data distribution. Thus, the heuristic selects only variables that are large enough to amortize such costs.

The MApp heuristic considers to place data closer to threads or spread them over the machine memory banks, depending on the NUMA factor of the machine and in the characteristics of the application. From MAi-array interface, we selected the *mai_ cyclic* and the *mai_ bind_ block* (block is a set of rows or columns) memory policies for the heuristic implementation. These memory policies were selected because they allow to place data closer to threads (*mai_ bind_ block*) or to spread data over the machine memory banks (*mai_ cyclic*). The choice between these two memory policies relies in the MApp heuristic described in chapter 4. The output generated by the heuristic is a text file with a list of variables, their information and their memory policies. This text file is later used by MApp to transform the application source code, in order to include the MAi memory policies for the variables.

5.2.4.2 Design a New Memory Policy

Although MAi has many memory policies, other applications and NUMA machines may have special needs. Therefore, specialized memory policies must be designed for such situations. One of the features of MAi is the support to design new memory policies and plug them in the array based interface. In this section we present a simple example of how to use MAi system functions to create a novel memory policy.

In order to design a memory policy, developers only have to write some functions

that specify how data will be distributed among the NUMA machine memory banks. These new functions must be written using MAi system functions such as architecture information functions and data distribution functions. The first set of functions allows developers to better understand the target platform and consequently better distribute data. The second one provides different ways to divide/cut arrays into blocks and to bind such blocks on memory banks.

```
int compute_neighbor(int nodesrc)
{
  int i,j,min=MAX;
  int node=-1;
  float nf; //numa factor

//Get NUMA factor between each pair of nodes
//and find the smallest one
  for(i=0;i<nnodes-1;i++)
    if(i != nodesrc){
      nf = na_get_numafactor(nodesrc,i);
      if(min > nf){
           min = nf;
           node = i;}
       }

  return node;
}

/*
void *array: pointer to data
pid_t id: id of the thread
*/
void my_distribution(void *array,pid_t id)
{
  int nodes[2];

//Get the node where the thread id is running
//and find its closest node neighbor
   nodes[0] = na_get_nodeidcpu(na_get_core(id));
   nodes[1] = compute_neighbor(nodes[0]);

//Set nodes to be used in data placement
   mai_nodes(2,nodes);

//Set array chunk to bind on the memory banks
  mai_bytes(array,mai_getsize(array)/2);

//Perform N binds, where N is the number of nodes
  mai_regularbind(array);
 }
}
```

Figure 5.11: Example of Development of a Memory Policy

In the example (Figure 5.11), we present an implementation of a new memory policy named *my_ distribution()*. This memory policy distributes data among memory banks in a regular fashion (regular chunks of data for each memory bank). Thus, all we have to do is divide data into blocks of the same size and then distribute such blocks on some memory banks considering the NUMA factor between then.

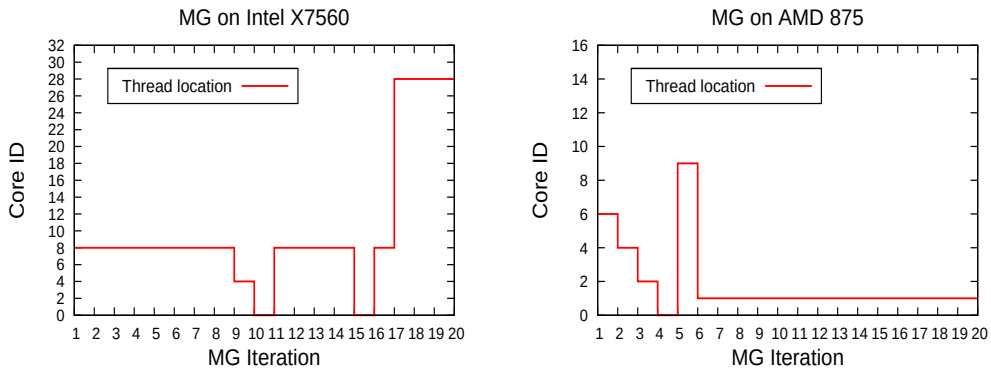
In function *my_ distribution()*, the first thing to do is to compute the memory banks that will be used to place data. The functions *mai_ get_ nodeidcpu()*, *na_ get_ numafactor()* and *mai_ nodes()* allow developers to obtain the node id that a core belongs to, to get the NUMA factor between two nodes and to set the memory banks that will be used in the placement. After that, developers must specify how arrays will be divided by using *mai_ bytes()* or *mai_ subarray()*. These two functions divide the array into blocks of the same size. In this example, the array was divided into blocks of same number of bytes, considering number of nodes used in the data distribution (two, in the example). Finally, we tell MAi that we want to bind these blocks on the selected nodes. In order to do this, developers just have to call the *mai_ regularbind()* function. This function will perform data alignment, data placement, compute available memory on the selected memory banks and make data migration (if necessary). In the case of the selected NUMA node does not have enough memory the function *mai_ regularbind()* returns an error to the application, indicating that there is not enough memory.

5.2.5 Mapping Threads to Enhance Data Locality

On a multi-core platform, the operating system (OS) usually distributes different processes/threads on all available cores in a way which allows the system to work most efficiently. The strategy used to perform such a distribution is named the thread affinity or thread placement. There are two ways of dealing with thread affinity in such platforms. The first way, called soft affinity, is to let the OS scheduler control the processes/threads migration. Usually, the OS scheduler will try to keep processes/threads on the same core as long as possible. However, in some situations the OS scheduler may migrate processes/threads to another core, even when it is not necessary, impacting on the overall performance of the system. The second way, called hard affinity, is to delegate the processes/threads locality to the user or runtime. In this case, the user/runtime may define on which core each thread must run.

Considering Linux operating system, we have observed how it schedules threads on a multi-core machine with NUMA characteristics for the MG benchmark from NAS Parallel Benchmarks. The selected benchmark is composed by a sequence of parallel iterations. For this experiment, we have traced the locality of all threads at the beginning of each iteration. The analysis of the trace obtained during the execution of the benchmark with one thread per core allowed us to conclude that the scheduler has changed constantly the locality of the threads.

To demonstrate such behavior, we obtained the trace information for a thread during the benchmark execution on two different NUMA platforms (Figure 5.12).



(a) (b)

Figure 5.12: Linux Scheduling for the MG Benchmark on NUMA Platforms.

The Intel X7560 machine has 32 cores and four NUMA nodes. Each node is composed by eight cores. The second machine is an AMD 875 with 16 cores and eight NUMA nodes. Each node of this machine has two cores. Table 5.3 specifies the physical core ids on each node. One can observe in Fig. 5.12 that the Linux scheduler has considerably varied the locality of the thread: on a different core (y axes) on the same processor, on different processors, on different NUMA nodes of the machines. All other threads have presented analogous behavior.

Since memory affinity is a relationship between threads and data, it is also important to place threads over the machine cores in order to improve memory affinity for a parallel application. Therefore, in the design of Minas framework we also included some support for thread placement. Particularly, each time that the OS decides to migrate a thread to a different core its data will remains on the previous cache and memory bank. On NUMA machines this can generate several remote memory accesses and may create contention. Thus, it is important to have data close to threads by migrating them. However, on Linux systems this can be a very expensive operation [Brice Goglin 2009]. In order to avoid this, in MAi, thread placement is managed by pinning threads to machine cores when they are created. This strategy avoids any thread migration by the operating system assuring better cache and memory usage.

Minas MAi runtime pins threads using the system call *sched_setaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask)* that allows MAi to specify a core for a thread. This system call receives as parameters the thread id (pid_t pid), the size of a cpu mask (number of bit necessary to index the machine cores) and the mask that specifies the core that must be used. The choice of which core to use for a thread is based on the machine topology. Before deciding which core to use, MAi call some numArch functions (e.g. *na_get_sharedCache()*) to discover how the cores share cache and main memory. After this, MAi pins threads considering cache hierarchy between cores to enhance or not cache sharing among threads of

Table 5.3: Intel and AMD Machines Topology.

	Node	Cores
Intel	#1	0,4,8,12,16,24,28
	#2	1,5,9,13,17,25,29
	#3	2,6,10,14,18,26,30
	#4	3,7,11,15,19,27,31
AMD	#1	0,1
	#2	2,3
	#3	4,5
	#4	6,7
	#5	8,9
	#6	10,11
	#7	12,13
	#8	14,15

a parallel region. MAi implements two classical thread placement strategies the compact and the scatter ones. In the compact strategy the goal is to enhance cache sharing among threads of a parallel region whereas, in the scatter strategy threads are placed on the machine in such a way that cache sharing among them is reduced. This default behavior can be changed by the user, providing a configuration file to MAi with the cores that must be used in the application execution.

Another possible way to change such behavior is the use of memory traces of the application to generate the configuration file. One can use memory access metrics such as the amount of shared memory and the access performed in the shared memory to identify the data sharing pattern between threads. Then, this trace can be used to group threads that work in the same data set. In the work [Cruz 2011], we have worked with Eduardo Cruz and Marco Zata from University Federal of Rio Grande do Sul to generate such configuration file. We have used memory traces to find the best suited thread mapping for NUMA machines. These memory traces have been used as input for Minas framework to pin threads to the cores of a machine.

5.3 Summary

In this chapter, we presented the Minas Framework components and its implementation details. This framework has been designed to manage memory affinity in parallel applications. It relies on three components that provide tools for allocating and placing data efficiently and in a portable way over NUMA machines.

The numArch extracts the NUMA machine topology and memory subsystem information, in order to provide a general and portable representation of the target NUMA machine. The numArch also provides a user-space functions that can be used to better understand a NUMA machine.

In order to reduce the complexity of memory affinity management and to provide a portable memory affinity solution, Minas MApp module extracts some information of the application in order to produce a NUMA-aware application source codes taking into account the NUMA machine characteristics. The last component of Minas framework has an important role, since it is responsible for all data allocation and placement. In order to provide a flexible and efficient implementation for MAi, we designed it with two levels. One is specific for arrays while the second is more general for dynamic data structures such as queues and trees. This design allows MAi to support different levels of memory affinity control (e.g. heap and variables). Furthermore, MAi implements several memory policies to deal with both regular applications and irregular applications.

The framework introduced in this chapter provides different supports to deal with memory affinity on parallel applications. Its components are basic tools that can be later used in a number of parallel programming interfaces, libraries and languages to control thread and data placement on NUMA machines. In the next chapter, we present how to use Minas framework to enhance memory affinity on parallel environments that do not support NUMA machines.

Chapter 6

Employing Minas Framework on Parallel Environments

Several popular parallel languages are available to program multi-core platforms with shared memory design. However, as presented in chapter 3, most of them do not have any memory affinity support. As a consequence, their performance may be reduced in NUMA platforms due to the non-uniform memory accesses costs. For instance, OpenMP [OpenMP 2011], Charm++/AMPI [Kalé 2009a] and OpenSkel [Góes 2010b] are examples of programming environments that do not have NUMA support. In this chapter, we present how to make use of Minas components on parallel environments for High Performance Computing to enhance memory affinity. To show the applicability of Minas memory affinity approaches and how they can be used to improve the performance of parallel applications on NUMA platforms, we select four different parallel environments, OpenMP, Charm++, AMPI and OpenSkel. We chose these environments because they lack NUMA support, they present different programming models and they can provide information of the application at compile time or runtime. We first present each parallel programming environment, describing their main characteristics. After that, we describe implementation drawbacks of each parallel environment considering NUMA machines. In light of these issues, we show how to make use of the Minas memory affinity approaches that address these problems.

6.1 OpenMP API

OpenMP is an API (Application Programming Interface) that provides a simple way to develop parallel applications for shared memory machines. The code parallelization is done using directives in the sequential code to parallelize loops or create tasks that are processed by a team of worker threads at runtime. These directives provide the compiler with information of which regions of the application have to be parallelized. Thus, all low level parallelization work relies on the compiler that uses the OpenMP macros on the code to generate the parallel application [OpenMP 2011].

The execution model used by OpenMP to run a parallel application is the fork-join one. In this model a master thread creates a team of worker threads when a parallel region is reached. The team of threads and the master thread work together in the parallel region to accomplish the work. At the end of the parallel region they synchronize. Figure 6.1 shows the schema that represents this execution model.

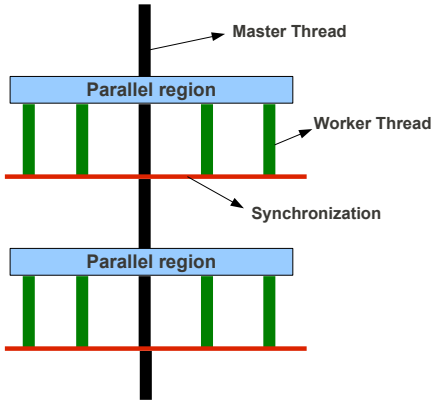


Figure 6.1: OpenMP Execution Model.

In this model, for every parallel region the master thread and the worker threads share a global shared memory space, which is composed by the shared variables. In OpenMP, variables are shared by default. Any private variable must be explicit set as private for an parallel region. Threads use these shared memory to communicate, hiding from the programmer the data transfer between them. Additionally, the API also specifies work sharing and synchronization mechanisms that can be used by the programmer to better compose a parallel application. The work sharing in OpenMP specifies how work is distributed among threads. For instance, the static work sharing describes a static work distribution, whereas in the dynamic one, work is distributed among threads at execution time. The synchronization mechanisms in OpenMP allows users to specify barriers and critical regions for parallel regions.

Although OpenMP has been developed for shared memory machines, it does not take into account that the target machine may have a hierarchical shared memory subsystem. The OpenMP standard does not have any memory affinity support on its specification. In fact, memory affinity on OpenMP relies on the operating system, which performs data and thread placement for the OpenMP parallel application. However, the operating system affinity support lacks in providing good performance for an OpenMP application, since it does not consider the OpenMP execution model to control memory affinity. Therefore, OpenMP fails in attaining scalable performance on multi-core machines with NUMA design.

6.1.1 Memory Affinity: Automatic management

We propose a transparent memory affinity support for OpenMP, which does not require explicit changes in the application source code, neither in the OpenMP interface and runtime. The automatic memory affinity support for OpenMP applications relies on the idea of using the application information extracted by the compiler and the architecture design to tune the application. Therefore, all components of Minas framework, the numArch, the MAi and the MApp are used to control memory

affinity for the OpenMP applications. However, since OpenMP applications are generally implemented using arrays, the MAi interface used for such applications is the MAi-array.

The numArch is actually used by MAi-array and MApp to retrieve the information of the NUMA machine without the interference of the programmer. This means that architecture abstraction for OpenMP applications is guaranteed by numArch. Using the information of the architecture, MAi-array is able to place thread and data over the NUMA nodes using any of its thread placement strategy and memory policies presented in chapter 5.

The choice of which memory policy must be applied for an array relies on the MApp preprocessor. The MApp scan the OpenMP application at compile time to extract information the application such as the variables and their memory access patterns. This preprocessor then matches the variable information with the NUMA platform characteristics to produce automatically a NUMA-aware OpenMP code. MApp produces this code by including functions from MAi-array interface inside the application source code to allocate and place data. The decision of which memory policy to use for each array of the application is made using the heuristic introduced in chapter 4, section 4.4.1. The final source code can be compiled with any compiler that has OpenMP support.

Although MApp considers the application and the machine characteristics to manage memory affinity, it may sometimes not produce the best performance application source code for a NUMA machine because it uses an heuristic. Due to this fact, we also let the programmer use the MAi-array interface to manually change the application source code.

6.1.2 Design and Implementation of Memory Affinity Support

The design and implementation of Minas framework for OpenMP relies on compile time implementation without the need of changing the compiler. Therefore, the Minas framework integration with OpenMP is done before the application execution in a static way. To do so, Minas includes MAi functions in the source code at compile time.

The first issue that we address in the designing of the memory affinity support for OpenMP is how to include Minas framework in the OpenMP compile process. To do so, we have implemented an user space tool that receives as a parameter the OpenMP application source code (i.e. the file that contains the main function) and then, it calls the MApp preprocessor to start the memory affinity management process. MApp is the only component of Minas that has been designed for a target parallel language, C with OpenMP. Additionally, MApp was designed for applications that are based on static arrays, no dynamic allocation in the source code is supported. However, its ideas and concepts can be easily adapted for other parallel languages.

MApp starts by calling CUIA to extract the variable information and the OpenMP constructions. It uses the OpenMP constructions for C language to retrieve the work sharing (*schedule* clause) for a parallel loop and the sharing type for variables of an

application. For instance, from the *schedule* clause of a *parallel for*, the parser extract the iterations for each available thread. Besides the *schedule* clause, the parser also uses the *private* and *shared* clauses to identify the sharing type for each application variable. Both *private* and *shared* are used in OpenMP to set the variables sharing type. In order to do that, the MApp parser looks for these clauses in all parallel regions defined by *pragma omp parallel* of the considered application. After the parsing, MApp uses its heuristic to decide the memory affinity strategy suited to the application variables.

The heuristic is used on OpenMP parallel loops (i.e. *pragma omp for*) in the original application source code. Consequently, the same variable may have different memory policies applied to it on the final application source code. This depends on the variable memory access patterns on each parallel loop. The application source code is then changed by the Minas source code transformation module with a set of functions from MAi-array interface. In the application execution, the Minas uses these functions to place thread and data over the NUMA machine nodes. To bind threads for OpenMP applications, Minas pins the threads created in a parallel region using the system call *sched_ setaffinity pid_ t pid, size_ t cpusetsize, const cpu_ set_ t *cpuset)*, introduced in the previous chapter. Minas perform this system call within a parallel region, in order to call it to each thread.

6.1.3 Illustrating Minas Framework with an Example

In previous sections, we have described how Minas framework can be employed in OpenMP to improve memory affinity for parallel applications. In this section, we present a complete example of how to use Minas to manage memory affinity on an OpenMP application. We select a snippet of a source code presented in Figure 6.2(a) as our target application. Additionally, we consider that the programmer does not know the application characteristics and starts the process using the Minas MApp to retrieve which variables are important for the application.

The list of important variables for memory affinity management is presented in Figure 6.2(b). Based in this list, MApp decides which memory policy is more suited for each variable using the heuristic described in the chapter 4. Then, the MApp preprocessor calls its source code transformation module. The transformation process is divided in three steps: include libraries, change variables declaration and include memory policies.

The first step of the transformation process includes the necessary libraries in the main file of the application. The included libraries are **mai_array.h** (for MAi interface functions) and **numa.h** (for Linux NUMA system calls). In the second step, all the static declarations of variables that are considered to be modified by MApp are changed to dynamic. The third step modifies the application source code in order to apply the selected memory policies for each variable.

Figure 6.3 shows the final code transformation generated by MApp. We can observe that MApp has only applied memory policies to two arrays, **M** and **Minv** (bold functions in Figure 6.3(b)). Differently, the array **I** was not considered because

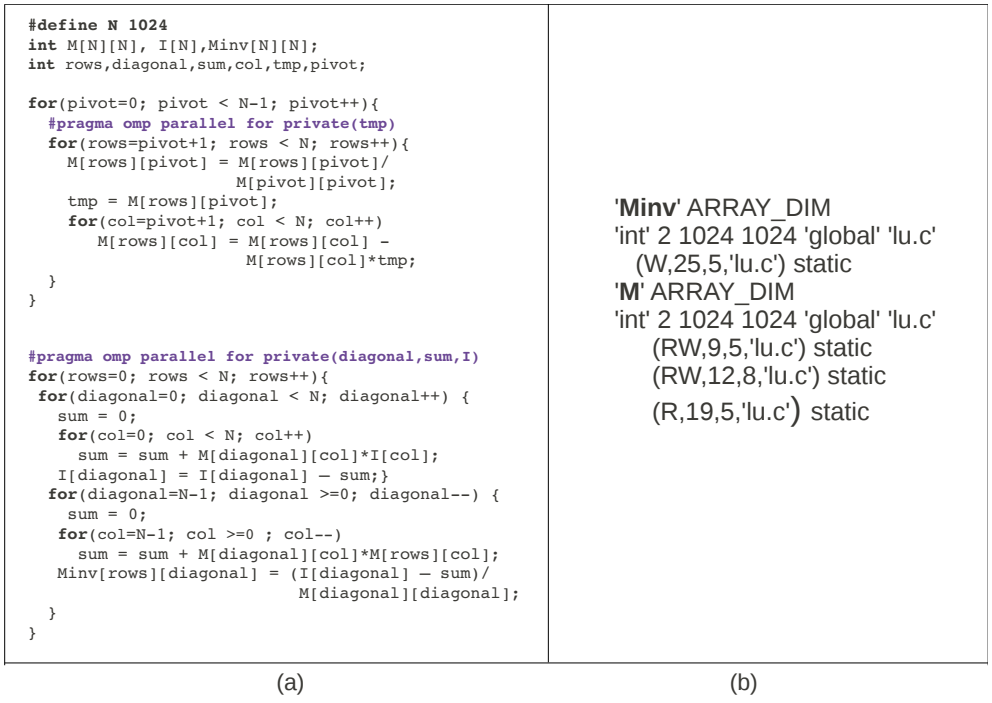


Figure 6.2: OpenMP Application and List of Variables Selected by MApp.

it is private to the second OpenMP parallel for loop (clause *private(diagonal,sum,I)* in the figure). Additionally, small variables such as *rows*, *diagonal*, *column*, *pivot*, *sum* and *tmp* probably fit in cache. Therefore, MApp does not interfere in the compiler decisions (allocation and placement of variables).

In this example, the target NUMA platform has a small NUMA factor among nodes. Therefore, remote accesses do not present high latencies. In the beginning of the application, MApp has applied *mai_bind_rows* memory policy for arrays **M** and **Minv**, since they are first used in write operations. In this case, it is important to place data close to threads that use them in order to avoid additional remote accesses. However, in the second phase of the application, **M** is used as read variable and may be a point of memory contention. Due to this, MApp has decided to spread memory pages of **M** with *mai_cyclic* memory policy in order to optimize bandwidth for the second phase of the application. In this case, during the application execution the memory pages that compose **M** are migrated to the new memory banks, when the *mai_cyclic* memory policy is called.

To provide an idea of how MAi places threads and data for this application at runtime, we present an example in Figure 6.4. We considered for this example a NUMA machine with four nodes and sixteen cores. The Fig. 6.4 (a) depicts the MAi thread and data placement for the first parallel loop whereas the Fig. 6.4 (b) shows the placement for the second one. The MAi function responsible for thread placement is the *mai_init()*. This function creates a parallel region with

```
int M[N][N], I[N],Minv[N][N];
int rows,diagonal,sum,col,tmp,pivot;


for(pivot=0; pivot < N-1; pivot++){
  #pragma omp parallel for private(tmp)
  for(rows=pivot+1; rows < N; rows++){
    M[rows][pivot] = M[rows][pivot]/
                     M[pivot][pivot];
    tmp = M[rows][pivot];
    for(col=pivot+1; col < N; col++)
       M[rows][col] = M[rows][col] -
                      M[rows][col]*tmp;
  }
}


#pragma omp parallel for private(diagonal,sum,I)
for(rows=0; rows < N; rows++){
 for(diagonal=0; diagonal < N; diagonal++) {
   sum = 0;
   for(col=0; col < N; col++)
     sum = sum + M[diagonal][col]*I[col];
   I[diagonal] = I[diagonal] - sum;}
  for(diagonal=N-1; diagonal >=0; diagonal--) {
    sum = 0;
   for(col=N-1; col >=0 ; col--)
     sum = sum + M[diagonal][col]*M[rows][col];
   Minv[rows][diagonal] = (I[diagonal] - sum)/
                          M[diagonal][diagonal];
  }
}
```

(a)

```
int **M, I[N],**Minv;
int rows,diagonal,sum,col,tmp,pivot;

mai_init(NULL);
M = mai_alloc_2D(N,N,sizeof(int),INT);
Minv = mai_alloc_2D(N,N,sizeof(int),INT);

mai_bind_rows(Minv);
mai_bind_rows(M);

for(pivot=0; pivot < N-1; pivot++){
  #pragma omp parallel for private(tmp)
  for(rows=pivot+1; rows < N; rows++){
    M[rows][pivot] = M[rows][pivot]/
                     M[pivot][pivot];
    tmp = M[rows][pivot];
    for(col=pivot+1; col < N; col++)
       M[rows][col] = M[rows][col] -
                      M[rows][col]*tmp;}}
mai_cyclic(M);

#pragma omp parallel for private(diagonal,sum,I)
for(rows=0; rows < N; rows++){
 for(diagonal=0; diagonal < N; diagonal++) {
   sum = 0;
   for(col=0; col < N;col++)
     sum = sum +M[diagonal][col]*I[col];
   I[diagonal] = I[diagonal] - sum;}
  for(diagonal=N-1; diagonal >=0; diagonal--){
    sum = 0;
   for(col=N-1; col >=0 ; col--)
     sum = sum + M[diagonal][col]*M[rows][col];
   Minv[rows][diagonal] = (I[diagonal] - sum)/
                          M[diagonal][diagonal];}}

mai_final();
```

(b)

Figure 6.3: Example of MApp source code transformation.

sched_ setaffinity() function in order to pin threads in the machine cores. In the Fig. 6.4 (a), one can observe that threads and their data are placed in the same NUMA node (same colors), due to the *mai_ bind_ rows* memory policy. Additionally, **M** and **Minv** arrays have been split by rows, each thread has a set of rows closer to it. However, in the second parallel loop the **M** array has been spread over the different memory banks. To do so, MAi has migrated the memory pages that compose **M**, to ensure the *mai_ cyclic* policy. For thread placement, MAi uses its default strategy that tries to place threads of a parallel region on cores that are closer, using one thread per core. We mean by closer, cores that share at least on cache memory level. In this example, since we have sixteen threads all cores are used by MAi.

6.2 Charm++/AMPI Parallel Programming System

Charm++ is a parallel programming system that has as main characteristic portability over platforms based on shared and distributed memory. It aims to provide a parallel programming support that abstracts architecture characteristics from the developer. To do so, this system provides two interfaces to develop parallel application, the Charm++ parallel C++ library and the AMPI (Adaptive Message Passing Interface) interface. Both rely on the concept of processor virtualization

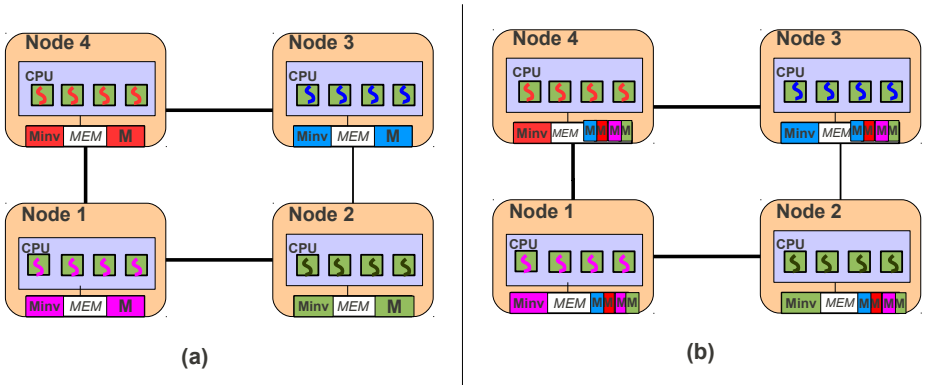


Figure 6.4: MAi Thread and Data placement.

technique. The processor virtualization provides programmers with support to write parallel application as a set of entities that represent the application computation. Charm++ parallel applications are written in C++ using an interface description language to describe its objects [Kalé 2009b] whereas AMPI parallel applications are written using MPI functions [Huang 2007].

In Charm++, computation is decomposed into objects named chares. The programmer describes the computation and communication in terms of how these chares interact and the Charm++ runtime takes care of all messages generated from these interactions. Chares communicate through a remote method invocation using a message-driven model. Further, the Charm++ runtime is responsible for physical resource management on the target machine. Figure 6.5 shows a schema that represents charm++ execution model. We can observe that chares A and C communicates with each other. In the pseudo code presented in the figure, this is done by the *C.entry(m)* procedure, which calls the chare C with the message created in chare A.

In the previous versions of Charm++ (before version 6.1), communication between chares was based on the exchange of messages. This means that for every communication, pack and unpack functions were used to create messages and, after that, messages were sent. This behavior was also implemented on shared memory machines, which may have a low performance. Because of this, Charm++ researchers have proposed in [Mei 2010] some optimizations for shared memory platforms.

In the current version of Charm++, all communication between chares on shared memory machines is done on memory. In the shared memory (SMP) build of Charm++, communication proceeds through the exchange of pointers between the threads. Due to this, the Charm++ runtime is able to avoid high overheads due to messages and reduce communication time. However, in the case of NUMA machines, this mechanism can be affected by asymmetric memory latency and bandwidth. Charm++ relies on the operating system memory affinity and does not explicitly control the placement of shared data in the memory.

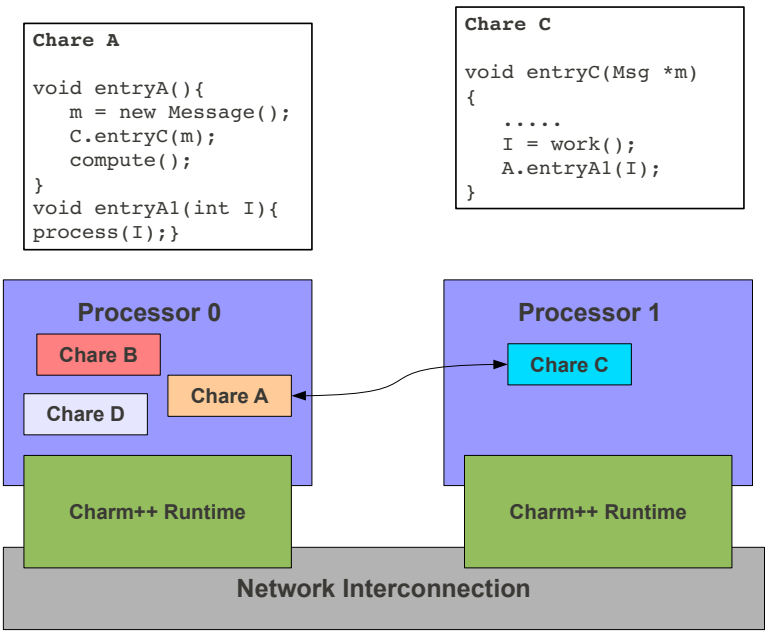


Figure 6.5: Charm++ Execution Model [PPL-Charm++ 2011].

Particularly, on some operating systems such as Linux and Windows, the default policy to manage memory affinity on NUMA machines is *first-touch*. This policy places data on the NUMA node that first accesses it [Joseph 2006, Carissimi 2007]. In the case of Charm++ communication mechanism, once the data (e.g. a message) is touched, this memory policy will not perform any data migration to enhance memory affinity.

This might result in sub-optimal data placement in Charm++ applications running on NUMA platforms. For instance, we can imagine a situation where some messages have been generated and originally allocated on core 0 of NUMA node 0. After that, these messages are sent to core 1 of NUMA node 1 and after several hops they end up on core N of NUMA node N. All these message sends are pointer exchanges of data that were originally allocated and touched in the memory of core 0. In such a scenario, several remote accesses will be generated for each communication.

Besides shared memory platforms, Charm++ also supports distributed memory platforms such as clusters of multi-core machines. This support is possible because of the AMPI interface of Charm++. AMPI is an adaptive MPI implementation that relies in Charm++ runtime system. In the AMPI, computation is represented as in normal MPI applications, decomposing the work between MPI processes and when necessary performing some communication between them. However, a MPI process in AMPI is represented as a set of virtual process. The virtual process is a user level threads that can migrate over the platform. In this way, AMPI can

provide a dynamic MPI process migration support for MPI based parallel applications [Huang 2007, Huang 2003]. AMPI has been built on top of the Charm++ system, which provides it with migratable objects that make possible the virtual MPI processes. These virtual processes are completely controlled by the runtime system.

In AMPI, when process migration must be performed, the runtime decides which objects have to be migrated to other processors. The main problem in this process is that for each thread migration, data allocated in the stack and heap must be also migrated with the thread. Additionally, any pointer used by the thread in the stack and heap space must also be updated after the thread migration. The Charm++ runtime system implements an efficient mechanism that avoid pointers update named Isomalloc [Antoniu 1999].

Isomalloc mechanism is based on a special memory allocator that uses a global unique virtual memory space to map data over different machines [Antoniu 1999]. The global unique memory space is implemented as a range of equal virtual address area on all processors. When a thread migrates from processor i to processor j, its data is just copied from processor i in the same virtual address in processor j. This mechanism avoids updating any pointers for the thread memory range, because the memory address in the new processor is the same of the old one. Since the number of pointers for each thread may be high (e.g function pointers and pointers of dynamic variables), Isomalloc reduces the overhead for the thread migration [Zheng 2006].

Although Isomalloc reduces the overhead for thread migration, it only deals with virtual memory, no physical memory is considered in its implementation. Since the current trend in high performance clusters is the use of multi-core machines with NUMA design, it is important to provide NUMA support in Isomalloc. On cluster of multi-core machines with NUMA design, the Isomalloc mechanism may not guarantee the best performances because it is not aware that the shared memory is actually physically distributed. Additionally, since the Charm++ runtime relies on the operating system memory policy to place data, it can not ensure that during the application execution threads will always have its own data closer to them.

In the case of Linux based NUMA multi-core machines, when Isomalloc copies data of a thread to the destination virtual memory the mapping between virtual to physical memory is done by Linux. In this case, Linux uses the *first-touch* which binds the Isomalloc virtual memory to the physical memory bank of the node that first asks for an address in the Isomalloc memory. In this context, it can perform mappings of memory pages to memory banks that are not in the same NUMA node of the thread that will compute it. Therefore, several remote memory accesses can be performed by threads, which may reduce the overall application performance.

6.2.1 Memory Policies to Enhance Memory Affinity on Charm++

The first NUMA support that we propose for Charm++ is the integration of Minas MAi interface on its runtime system [Ribeiro 2010d]. Minas MAi provides Charm++ programmers some memory policies to place data of an application on

a NUMA machine in a transparent way. This means that Charm++ developers do not need to modify their applications source codes. The Minas MAi interface in Charm++ is implemented as a command line tool that allows users to select the memory policy from a list of possible policies for an application execution. For instance, this command line tool is used as an extra parameter on the Charm++ application command line execution.

It is important to mention that for Charm++, we use Minas MAi memory policies applied to the data of a task (Charm++ threads). The Minas MAi interface binds the Charm++ application data on the NUMA nodes where the Charm++ threads are running, following the selected memory policy strategy. Considering Charm++ system characteristics, we employ three memory policies from Minas MAi in its runtime: *bind all, cyclic, cyclic neighbors*. Figure 6.6 shows the main differences between memory policies; colors are used to represent threads and their data.

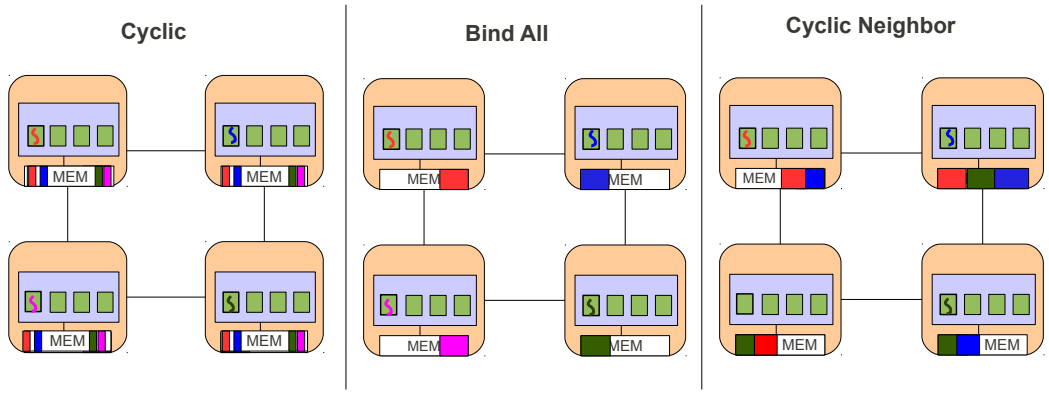


Figure 6.6: Memory Policies for Charm++.

The memory policy *bind all* binds all data of a task to a physical memory bank, in order to associate data of a task to a memory node. Such memory policy is more suited for applications, where the Charm++ thread allocates its own data and it only uses it during the application execution. This memory policy works similarly to *first-touch*, except that the Charm++ user can select which memory banks a task should use to place its data and not only the one in which it is running. In this way, users can for example exclude memory banks attached to nodes that performs I/O, which generally offer worse performance.

The *cyclic* memory policy distributes data over all NUMA nodes in a round-robin way whereas the *cyclic neighbors* also spread data over a subset of NUMA nodes of the machine. However, *cyclic neighbors* only use NUMA nodes that are neighbors of the node where the Charm++ task is running. In the case of Charm++, Minas does not pin its threads to the machine cores. Since Charm++ already have thread placement support, we use this Charm++ support to pin threads of an application.

6.2.1.1 Design and Implementation of Memory Policies

The first point that we must cover in the design of the memory affinity policies support for Charm++ is how to manage the interaction between Charm++ and Minas. To do so, we have included a memory affinity module in the *converse* layer of Charm++. Converse is a message passing layer that provides portability and messages management on Charm++ [Kalé 1996]. The memory affinity module inside converse allow us to call Minas to extract hardware information (numArch) from a NUMA machine and to set a memory policy (MAi) for each thread of Charm++ (except communication threads). Basically, this module receives from Charm++ runtime information about threads of an application and the memory policy selected by the user. Using such information and hardware characteristics, the memory affinity module applies the memory policy for each thread and synchronize all threads. After that, the application will start to run following the memory policy.

```
void CmiInitMemAffinity(char **argv) {
int policy = -1; char *mempol = NULL;

//parsing the selected memory policy
CmiGetArgStringDesc(&mempol); convertPol(policy,mempol);

//Where the caller thread is running
int myPhyRank = CpvAccess(myCPUAffToCore);

//call numArch to get the Node of the myPhyRank core
int myMemNid = na_nodeidcpu(myPhyRank);

//call numArch to get the number of neighbor of each node
int *myNgbId = malloc(na_get_numNeighbor()*sizeof(int));

if(policy == CYCLIC_NEIGHBOR) {
   //call numArch to get the node neighbors
   na_getneighbors(myMemNid,&myNgbId);

   //apply the memory policy, call MAi interface
   if(mai_cyclic_neighbor(myNgbId)<0)
        CmiAbort("mai_cyclic_neighbor error - Nodes Id");
   free(myNgbId);
}

 CmiNodeAllBarrier();
}
```

Figure 6.7: +maffinity Code Sniped.

In order to provide an interface between Charm++ users and Minas MAi module inside the converse core, we have implemented the command line option named **+maffinity**. This command line is composed of the selected memory policy and in the case of *bind all*, *cyclic* and *cyclic neighbor* memory policies, it also has the

NUMA nodes that must be used. In the beginning of the execution the command line is parsed by Charm++ system and then, Minas MAi memory policies are called considering the parameters selected by the user. In order to ensure good affinity for Charm++ applications, **+maffinity** must be used with **+setcpuaffinity**, since it avoids any thread scheduling by the operating system. Thus, ensuring that threads will not lose their memory affinity during the application execution.

In Figure 6.7, we present a snippet of code from the **+maffinity** module of Charm++ (*cyclic neighbor* memory policy). We can observe that the implementation of the memory policy also relies on some functions of Charm++ system (e.g. *CpvAccess()*, *CmiNodeAllBarrier()*). These functions allow Minas MAi to extract information of the current task to set the memory policy and to synchronize them before starting the application exection. For instance, the *CpvAccess()* is used to retrieve the core where a thread is running.

6.2.2 NUMA-Aware Load Balancer

Different approaches can be used to enhance the memory affinity on NUMA machines. Aside from memory allocation, memory policies and thread mapping mechanisms, load balancing mechanisms can also be used to improve memory affinity on NUMA machines.

Charm++ offers a load balancing database interface that is capable of providing important information about an application execution [Brunner 2000, Zheng 2005], such as statistics about the communication between the chares and their load. This information can later be used to improve the load balancing and to consequently enhance memory affinity on multi-core machines with NUMA design. However, Charm++ still lacks information about memory access costs and the machine topology, which represent important aspects of the NUMA platform. Using the Charm++ database interface and Minas numArch module, we propose a NUMA-aware load balancer for Charm++ named NumaLB [Pilla 2011a, Pilla 2011b].

The NumaLB load balancer relies on matching together in a single load balancer the characteristics of the application and of the NUMA machine to improve memory affinity. It is a List Scheduling, greedy algorithm, that picks the heaviest (longest execution time) unassigned chare and assigns it to the core that presents the smallest cost. The choice of a greedy algorithm is based on the idea of fast converging to a balanced situation by mapping the greater sources of unbalance first. Since the objective of NumaLB is to reduce communication overhead, it only migrates a chare to the closest processor. In order to evaluate which processor is the closest one, we propose the heuristic defined by the following equation:

$$cost(k,i) = L(i) + \alpha \times (-M(k,i) + \sum_{j!=i}^{nproc} (\mathrm{M(k,j)} \times \mathrm{NF(i,j)})) \qquad (6.1)$$

Where:
- cost(k,i) is the cost of migrating a chare k to processor i
- L(i) is the load of processor i

– M(k,i) is the number of messages exchanged between chare k and chares on processor i
– NF(i,j) is the NUMA factor from i to j

The heuristic relies on the Charm++ load balancing database interface to extract the CPU load ($L(i)$) and the chares communication graph. The CPU load allow us to have an overview of how processors are being used for the evaluated application whereas the chares communication graph provides an overview of how chares share data over the NUMA machine. In order to represent the NUMA machine hierarchy and topology, we use the NUMA factor (obtained with numArch) which provides a good estimation of the machine access latency to different nodes ($NF(i,j)$). The alpha in the heuristic equation is used to balance the different metrics used. For instance, the load of the processor is provided by Charm++ system in seconds and the communication is provided in number of messages.

For each chare not yet assigned to a processor, the heuristic evaluates the load of a processor and the communication between the analyzed chare and processor. If a processor is too heavy the chare will only be migrated to this processor if its communication with the processor is too important ($-M(k,i)$). Additionally, we also consider the communication that the chare has with all other processors ($M(k,j)$). This must be evaluated because when a chare migrates to a different processor its communication costs with all other chares will change ($\sum_{j!=i}^{nproc}$(M(k,j)$\times$ NF(i,j))). Therefore, they can have an important impact in the overall performance of the application. Finally, the smaller W is, greater is the possibility of k to migrate to processor i.

6.2.2.1 Design and Implementation of NumaLB

In Charm++, any load balancer is implemented using its load balancing interface. Therefore, the NumaLB load balancer is implemented as load balancer class that extends the CentralLB class of Charm++. The CentralLB provides the interface for the *init() - work(LDStats *stats)* methods. The *init()* method is responsible for the initialization of the NUMA machine information into data structures of NumaLB. The *work(LDStats *stats)* method is responsible for the NumaLB strategy, which performs the load balancing strategy.

In the *init()* method, the machine information used by NumaLB is the architecture topology and its interconnection network. The topology comprehends the organization of the NUMA nodes and the cores inside the machine whereas the interconnection network is represented by the NUMA factor table. This table is a square matrix composed of the NUMA factors for all pair of nodes of the machine. In order to get the machine topology, we use the functions *na_get_maxcpus()*, *na_get_maxnodes()* and *na_get_nodeidcpu(core)*, which provide respectively, the number of cores, the number of nodes and the node of each core. These information is loaded in private variables of NumaLB to avoid any unnecessary call to the Minas numArch library. In Figure 6.8, we can observe these private variables in the init() function.

```
void NumaLB::init() {
  lbname = (char*)"NumaLB"; alpha = _lb_args.alpha();

  int i,j; if (CkMyPe()==0){
     //if NUMA machine, load machine topology
    if(na_is_numa()){
      max_cpus = na_get_maxcpus(); max_nodes = na_get_maxnodes();

      //NUMA factor square matrix
      numaFactorMatrix = new float [max_nodes*max_nodes];
     for(i=0;i<max_nodes;i++) //fills NUMA factor matrix
       for(j=0;j<max_nodes;j++)
         numaFactorMatrix[i*max_nodes+j] = na_numafactor(i,j);

      //get the node for each cpu
      cpusToNodes = new int [max_cpus];
      for(i=0;i<max_cpus;i++)
       cpusToNodes[i] = na_get_nodeidcpu(i);
   }
 }
}
```

Figure 6.8: NumaLB Code Sniped.

In the Minas numArch library, the NUMA factor is computed using some benchmarks. This process can take some considerable time to finish, depending on the machine size. Therefore, in order to avoid this overhead to get the NUMA factor at execution time, we perform such computations at Charm++ installation. This information is then stored in text files that are latter loaded into data structures using the *na_numafactor(i,j)* function. This is done at the load balancer initialization.

The init() function seen in Figure 6.8 has an important role in the NumaLB, since it loads the NumaLB data structures that represents the NUMA machine topology and memory access costs. Information such as the numaFactorMatrix and the cpusToNodes matrices are loaded in the load balancer using the numArch functions *na_numafactor(i,j)* and *na_get_nodeidcpu*. Both matrices are later used by NumaLB strategy to compute the heuristic for each chare of the application.

In the NumaLB class, the *work()* method is implemented using the matrices created in the *init()* method and the application information provided by Charm++ through *LDStats *stats*. The LDStats provides to load balancer classes a data structure with the information of the application. Using this data structure the load balancer can retrieve the processor load and the chares communication graph. Basically, in the *work()*, NumaLB implements a for loop that goes through all chares of the application computing the costs to migrate them over the machine core (our heuristic). Then, the smallest cost is used to place a chare C in a processor P. The CentralLB also provides to load balancer classes methods to set the new processor for a chare (*setNewPe(best_proc)*). This is performed at the end of an iteration

of the NumaLB loop. When all chares have been distributed over the processors, the NumaLB sends the new mapping to the Charm++ system using the method *convertDecisions(stats)*, which tells Charm++ that the load balancing has been performed and that processors must start to work.

6.2.3 NUMA-aware Isomalloc Memory Allocator

Considering the AMPI system, we propose a NUMA-aware support for its Isomalloc Memory Allocator [Ribeiro 2010d]. The NUMA-aware Isomalloc is an optimization of the standard Isomalloc memory allocator that has been used in AMPI. Isomalloc was originally designed for clusters of shared memory machines with UMA characteristics. Our proposal considers the hierarchy of the memory subsystem of a NUMA machine to efficiently place data for each thread migration.

NUMA-aware Isomalloc uses the application runtime information (e.g. where threads are running and memory usage) and architecture characteristics (e.g. number of nodes and distance betwen nodes) to dynamically decide physical data placement. Its main characteristics are its transparent control of memory affinity for NUMA machines and its dynamic mechanism that allows it to be used on different applications and architectures.

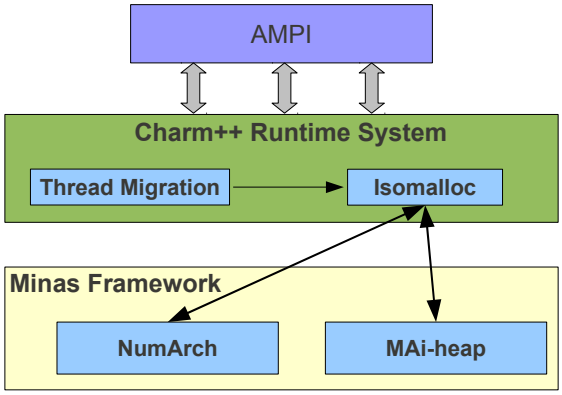


Figure 6.9: NUMA-aware Isomalloc.

The main challenges to implement the NUMA-aware Isomalloc mechanism is how to get the machine characteristics and to improve memory affinity dynamically for an AMPI application in a NUMA machine. To do so, we integrated Minas MAi and numArch modules inside the Charm++ runtime. Using the numArch and Minas MAi modules, the NUMA-aware Isomalloc mechanism is capable of extracting the machine topology at runtime and to apply a memory policy to the Isomalloc area for every thread migration. Figure 6.9 shows the interaction between Minas framework and the Charm++/AMPI runtime system. On every thread migration, the Isomalloc module is called by the thread migration to perform data migration for

a thread. After that, the Isomalloc calls numArch to require the machine topology and to apply the memory affinity for the data of a thread.

Considering the current NUMA machines and AMPI applications, we propose the integration of three memory policies from Minas MAi interface in the NUMA-aware isomalloc: *cyclic, cyclic neighbor* and *bind all*. However, inside the AMPI runtime they are known as *node cyclic, node neighbor* and *node affinity*. The only difference between these memory policies and the original ones on Minas MAi is the granularity used for the data placement. Since in the Isomalloc area each task has its own unique memory area, this area is the granularity used on memory policies. The choice of which memory policy to use relies on AMPI users by passing an additional parameter to the application execution command line, the memory policy name.

It is important to mention that for the AMPI, we do not pin threads to the machine cores. Since the idea in AMPI is to perform thread migration to reduce the execution time for an application, Minas does not make any thread placement. Instead, Minas relies on Charm++ runtime system to retrieve on which core a Charm++ thread is running. Using such information, Minas is able to migrate data for the thread.

6.2.3.1 Design and Implementation of NUMA Isomalloc

The implementation of the memory policies inside Isomalloc has been done in two distinct steps. A first one that extracts the machine information and a second one that bind virtual memory pages to memory banks following one of the memory policies. The first step is static and it is performed only once at Charm++ runtime initialization. The second step is dynamic and depends on the application execution characteristics.

In order to distribute data over the memory banks, the NUMA-aware Isomalloc must know the target machine characteristics such as the number of NUMA nodes, the NUMA factor and the machine topology. At the runtime system initialization, NUMA-aware Isomalloc extracts information about the target machine by calling Minas numArch module. After that, information such as number of nodes, core to node mapping and node distances are loaded into a data structure that represents the machine topology and hierarchy (the *numaTopo*). This data structure is later used by NUMA-aware Isomalloc at runtime to map memory pages into memory banks.

When the AMPI application is running and a virtual process migration has to be performed, the NUMA-aware Isomalloc algorithm retrieves from the data structure *numaTopo* the NUMA node where the virtual process is located. After that, it calls the Minas MAi functions to apply the selected memory policy for the Isomalloc memory range.

Considering the *node affinity* policy (Figure 6.10), Minas MAi starts by retrieving the memory banks where the memory range is mapped for a virtual process. This information allow Minas MAi to decide if it has to migrate data or not. If the virtual process memory range already respects the selected memory policy, no migration is

performed. On the contrary, it performs the data migration and places the memory range on the nodes specified by the memory policy (*mai_migrate()*).

```
void node_affinity(void *addr,int nr_pages) {
  int cpu,node,err,phys_node,actual_node;

  cpu = na_get_physcore();

  //get current node of the task
   phys_node = numaTopo[cpu].node;
  //search  node of the memory range
   actual_node = mai_get_node(addr);

   //nodes are different - we must migrate
      if(phys_node != actual_node){
       //the thread node has enough memory
        if(na_has_memory(phys_node))
          err = mai_migrate(addr,nr_pages,phys_node);
        else
         //search for neighbors to migrate data
         err = mai_migrate(addr,nr_pages,
               na_search_neighbors(phys_node,nr_pages));
    }
    else
     //just bind memory to the phys_node
      err = mai_bind_all(addr,nr_pages,phys_node);
}
```

Figure 6.10: Node Affinity Code Snipped.

Before applying any memory policy, Minas MAi always verifies if the destination node has enough memory. If it is not possible to place all of the memory range on the virtual process node, Minas MAi searches for the possibility to place such range in the neighboring nodes. In order to do so, Minas MAi uses a configuration file that describes the NUMA nodes interconnection network, provided by the numArch module, to find which nodes have the smallest communication costs (*na_search_neighbors()*). The ones with the smallest NUMA factors for a node i are considered by Minas MAi as neighbors of node i.

For the cyclic memory policies, Minas MAi always performs data migration to ensure the cyclic behavior for the physical allocation of the application data. The implementation of the cyclic memory policies are similar to the *node affinity* one. The difference is on the NUMA nodes that are used to perform data placement. The current version of the NUMA-aware Isomalloc supports only the Linux operating system because Minas MAi relies on the system call *mbind()* provided by NUMA API.

6.3 OpenSkel a Worklist Transactional Skeleton Framework

Parallel patterns allow programmers to implement parallel applications in a simple way. They are usually implemented as skeleton frameworks that hide from the user complex and frequently used communication and synchronization structures [MacDonald 2000]. To do so, these frameworks provide high level APIs that encapsulates the parallelism. In this case, programmers develop the parallel application as a sequential one. In this section, we introduce OpenSkel, a Worklist Transactional Skeleton Framework and its main features [Góes 2010a, Góes 2010b]. Then, we demonstrate how Minas framework can be used to provide a NUMA support for OpenSkel.

OpenSkel is a worklist transactional skeleton framework that has as main purpose simplify the development of parallel applications. It aims at providing an interface that combines transactional memory with skeletons in order to develop parallel applications that follow the worklist pattern. OpenSkel is composed of a runtime system and an API to handle transactional worklists. In this framework, parallel applications are written in C using its interface to describe shared and private data, work-units and transactional worklists [Góes 2010a, Góes 2010b].

In order to deal with transactional worklists, OpenSkel lets existing word-based STM (Software Transactional Memory) systems to control the transactions. In this way, programmers only have to describe what are their work-units, add them into a worklist and then, the OpenSkel runtime manages computation inside the worklist. All communication, synchronization and physical resources management (e.g. number of cores to use) are controlled by the OpenSkel runtime system. Figure 6.11 shows a schema that represents OpenSkel a worklist skeleton model and a OpenSkel pseudocode.

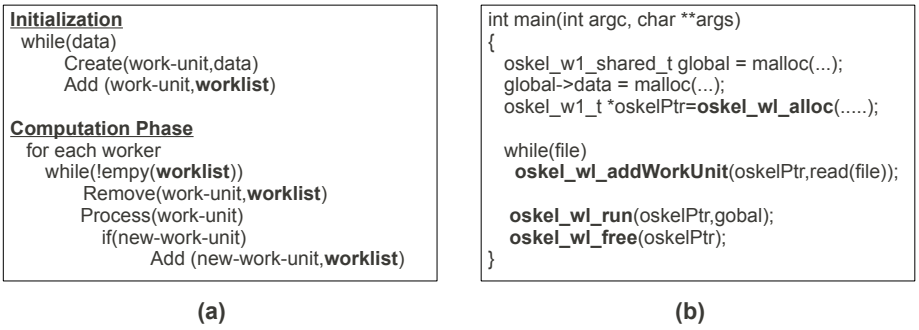


Figure 6.11: OpenSkel: (a) worklist skeleton model (b) pseudocode.

One can observe in Figure 6.11(b) the OpenSkel API functions to allocate (*oskel_wl_alloc()*), run (*oskel_wl_run()*) and free (*oskel_wl_free()*) a worklist. The API also provides a function to add work-units in the worklist (*oskel_wl_addWorkUnit()*).

The complete description of the API can be found in [Góes 2010a].

OpenSkel has been designed for shared memory platforms and in its current version all data allocation is performed by the master thread (i.e. work-units and worklist). Additionally, similar to Charm++ parallel system and AMPI, OpenSkel also relies to the operating system the control of all physical data placement. This design works well in shared memory machines with Uniform Memory Access characteristics, since memory access costs are similar to any of the machine cores. However, OpenSkel can have its performance decreased on shared memory machines with NUMA design, because data will reside in the master thread NUMA node. In this case, several concurrent remote accesses to the master thread node will be generated by every thread that want to access the OpenSkel worklist. Thus, poor performances for applications developed with OpenSkel API may be expected on NUMA machines.

6.3.1 Memory Affinity through Data Allocation and Memory Policies

In OpenSkel, a global shared worklist is split into chunks of work-units, and these chunks are processed by each thread of the application (similar to the OpenMP approach). However, the splitting strategy only passes pointers to threads. In the case of NUMA machines, memory pages that compose the worklist and work-units still remain on the master thread memory bank. In order to avoid NUMA impacts in this centralized worklist, we propose the integration of Minas memory affinity mechanisms in the OpenSkel runtime. For instance, a specialized memory allocation to the worklist and some memory policies to control data placement over NUMA machines.

The worklist on OpenSkel can be seen as an one dimensional array of work-units, where each work unit is one element of the array. The memory allocation mechanism must ensure that the memory zone reserved for the worklist will be used only by it. No other data from the application should be allocated in this memory zone. This approach ensures that the worklist will be continuous on virtual address space, allowing us to have a better control of its memory pages placement. Additionally, if no other data is in the same memory zone, we simplify the complexity and cost of data placement. In this case, we do not have to search for non work-units inside the allocated memory zone.

To enhance memory affinity in OpenSkel, the memory space that represents the worklist must be split distributed over the machine memory banks considering the access performed by threads on it. To split and place the worklist memory pages we consider the properties of the applications that are developed with OpenSkel. Such applications are based on the transactional memory model and their main characteristic is the high/low level of contention on data accesses. In order to support these two types of contentions, we use two memory policies: *bind_all* and *cyclic_neighbor*. The memory policy *bind_all* places memory pages of the worklist close to the thread that computes it whereas *cyclic_neighbor* spread them on the

nodes that are neighbor (1 hop in the network) to the node where the thread is running. Considering applications with low contention it is important to place data close to threads that uses it because the data sharing level is small. On the contrary, applications in which threads dispute data access it is important to guarantee good throughput.

The memory affinity support in OpenSkel using Minas framework also employs Minas thread placement mechanism to avoid any thread migration by the operating system. In this case, OpenSkel relies on Minas to retrieve the machine topology and pin threads over the machine cores. Two strategies can be used for OpenSkel, *compact* and *scatter*. The *compact* strategy enhances cache sharing between threads of the same team whereas. On STM applications with a high level of data sharing, this strategys thread placement reduces the communication costs between them. Additionally, on NUMA machines it allows Minas to reduce the number of remote accesses, since Minas can pin threads to cores of a same NUMA node. The *scatter* strategy reduces cache sharing between threads of a team. In the STM applications with a low level of data sharing between threads, this is the strategy used by Minas to place threads on the machine cores since it provides more cache memory to threads.

6.3.2 Design and Implementation Details

The design and implementation of memory allocation and data placement inside OpenSkel relies on the integration of Minas MAi and numArch inside its runtime system. Any memory allocation inside the OpenSkel runtime system uses the MAi-heap interface. Therefore, memory allocation of worklist relies on the *mai_ alloc(size_ t bytes)* function. Using the *mai_ alloc(size_ t bytes)*, MAi reserves continuous virtual address space with page alignment for the worklist of OpenSkel, in order to have a separate heap for the worklist.

Considering memory policies to place OpenSkel data on memory banks, we modified the original *mai_ bind_ all* and *mai_ cyclic_ neighbor* memory policies of MAi to include the information of the OpenSkel thread that is calling the memory policy. Thus, the two functions used to perform the data distribution are *mai_ bind_ local(tid id, int mycpu)* and *mai_ cyclic_ neighbor(tid id,int mycpu)*. The *id* is used by MAi to retrieve the thread that is calling the memory policy. The *mycpu* is used by MAi to get the node where the worker thread is running. To use the memory policies, some functions from numArch module are also needed to get information about the target machine and the application state (e.g. node where a thread is running). We also use the function *mai_ init(void *wl, int chunk)* to provide Minas with the information about the worklist (pointer) and the chunk size (selected by the user).

6.4 Summary

In this chapter, the main objective was to show the applicability of Minas framework to manage memory affinity on parallel programming environments. We pre-

sented how Minas framework components can be used or integrated into runtimes to manage data placement on parallel interfaces that are not NUMA-aware. We also described how Minas framework is used in each of the parallel interfaces, providing some implementation details.

Considering the OpenMP parallel interface, we have shown that users can employ all Minas framework components in parallel applications. The structural way of OpenMP constructions (e.g. pragmas) allow us to exploit the interface information inside Minas components to manage memory affinity. Besides its constructions, the OpenMP clauses provides Minas with the necessary information of how the work will be distributed over the worker threads and how these threads will access data.

In the case of Charm++ and AMPI, both interfaces can be used to develop parallel applications for cluster of NUMA machines. Due to this, we have integrated Minas MAi and numArch components in their runtime systems to deal with memory affinity. In Charm++, we used the Minas components to implement memory policies and a NUMA-aware load balancer. These mechanisms allow Charm++ to deal with data placement and thread placement over NUMA machines. For AMPI, Minas components were integrated to support a NUMA-aware Isomalloc memory allocator that extends the efficient AMPI thread migration to clusters of NUMA machines.

For the OpenSkel framework, we have integrated the Minas MAi and the numArch components inside its runtime. This integration provides the necessary support to deal with thread and data placement in NUMA platforms. Application information is extracted by OpenSkel and passed to Minas framework in order to efficiently manage memory affinity for parallel applications. Considering OpenSkel characteristics, we have used the MAi-heap interface, since applications developed with this framework generally uses dynamic data structures.

In the next chapters, we present the performance evaluation of Minas framework using parallel benchmarks and applications developed with OpenMP, Charm++, AMPI and OpenSkel.

Part III

Performance Evaluation: Case Studies

Chapter 7

Experimental Methodology

In this chapter, we present the experimental methodology used in the performance evaluation of Minas framework. We first describe the NUMA multi-core platforms in section 7.1. After that, in section 7.2, we depict the software stack used on each platform. In section 7.3, we introduce the selected metrics to evaluate the performance of Minas. Finally, in section 7.4 we present the measurement methodology used in our experiments.

7.1 NUMA Multi-core Platforms

In order to conduct our experiments and evaluate the performance of Minas framework, we have selected three representative NUMA multi-core platforms. These machines are representatives because they have different NUMA characteristics. For instance, the cache coherence protocols, the interconnection network and the architecture organization. In this section, we introduce these three platforms describing their main characteristics and their architectural design differences.

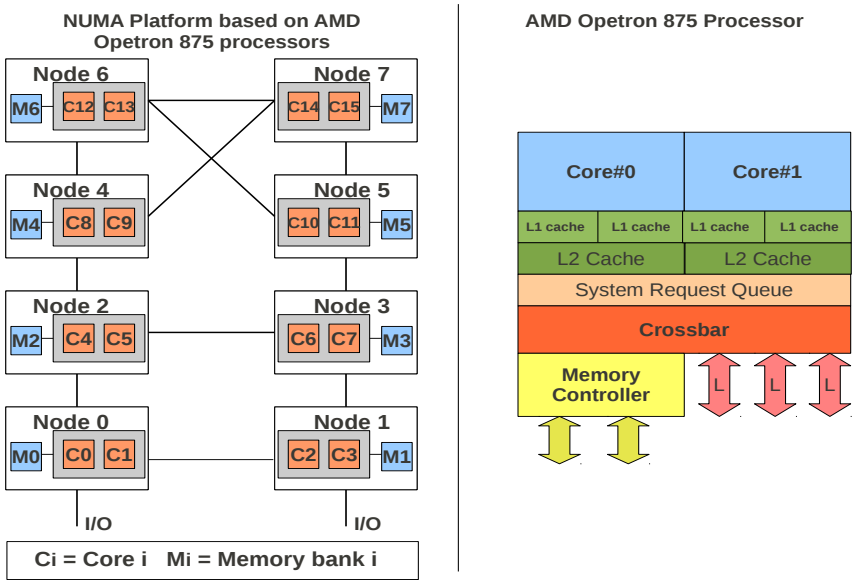


Figure 7.1: NUMA based on AMD Opteron Processor.

The first platform is composed by eight Dual Core AMD Opteron Processor

875 (2.2 GHz). This platform is organized in eight nodes of two cores, each one with private caches L1 and L2. It has a total of 32 GB of main memory, 4 GB of local memory in each NUMA node. Each node has three interconnections (HyperTransport) to other nodes, except for nodes zero and one that have only two. This exception is because the third interconnection on nodes zero and one is used for input and output devices. A schematic representation of this machine and of its processor is given in Figure 7.1. We can observe that this platform does not have any shared cache memory, each core has two levels of private cache memories.

The second platform relies on the Intel Xeon X7460 (2.66 GHz) processors with six cores each. The machine is composed of sixteen processors which are organized in four nodes, each one with four processors. Therefore, each NUMA node has twenty four cores. There is a shared cache L3 of 16 MB per processor and shared cache L4 of 256 MB per node on this machine. It has a total of 192 GB of main memory (48 GB of local memory) that is physically distributed over the machine nodes. Each node has three point-to point interconnections (Front-Side Bus) to other nodes. A schematic representation of the machine is given in Figure 7.2. Every access to the global shared memory is controlled by a memory controller. Each node of the machine has a memory controller that is shared by four processors. Due to this, some contention and long access times are expected in this platform.

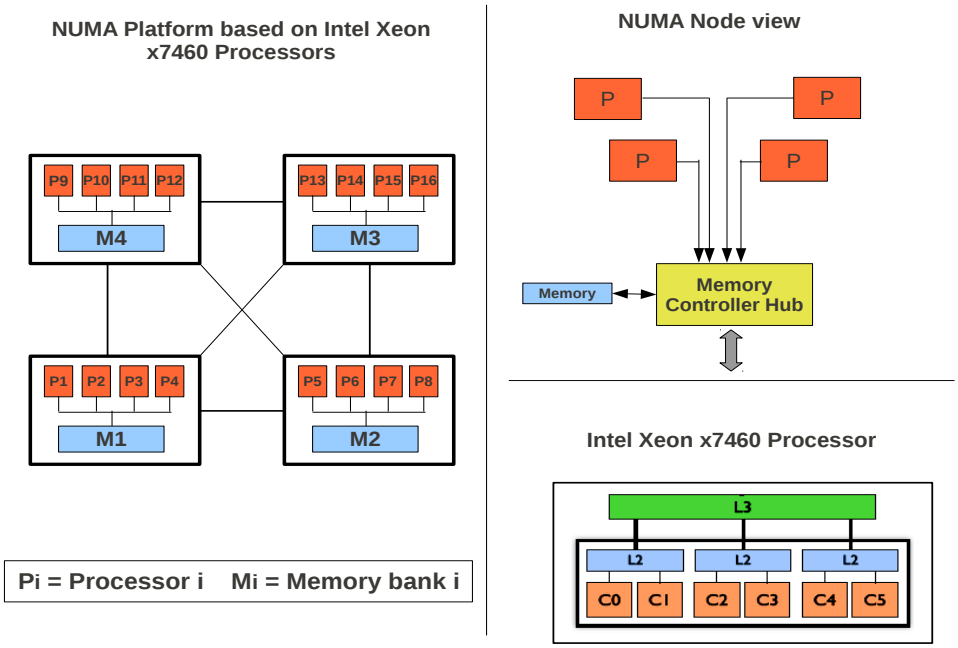


Figure 7.2: NUMA Platform based on Intel Xeon X7460.

Finally, the third platform used in this work is a machine composed of four Intel Xeon X7560 (2.27 GHz) processors with eight cores each. There are four nodes in

total, being one processor per node. Each core has a private cache L1 (32Kbytes for data and 32Kbytes for instruction) and L2 (256Kbytes). Additionally, there is a shared cache L3 of 24 MB per processor. This cache memory hierarchy reduces communication costs between cores of a node. The platform has a total of 64 GB of main memory, each node has 16 GB of local memory. On each node, there are three point-to-point QPI (Quick Path Interconnection) interconnections to other nodes. A schematic representation of the platform and of its processor is given in Figure 7.3. In this machine, each processor has an integrated memory controller that manage all memory requests.

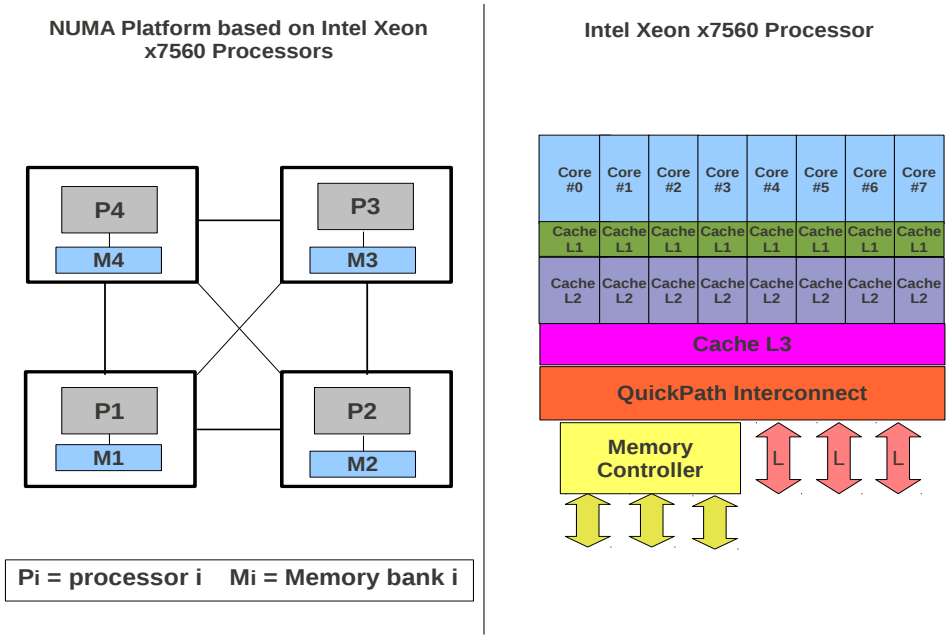


Figure 7.3: NUMA Platform based on Intel Xeon X7560.

The main differences between the three platforms presented above are: (i) their interconnection network, (ii) cache coherence protocol and (iii) NUMA factor. These characteristics have an important impact on memory access costs, since they are responsible for the number of hops, number of messages and latency for each memory operation. In the rest of this document, the machines are named using a code convention, composed by the processor manufacturer, the number of NUMA nodes and the number of cores per node. The first machine is thus named AMD8x2, the second one Intel4x24 and the third one Intel4x8.

Table 7.1 summarizes the characteristics of each machine. Main memory bandwidth (obtained from Stream Benchmark [Mccalpin 1995]) and NUMA factor (obtained from BenchIT [The BenchIT Project 2010]) are also reported in this table. The main memory bandwidth is obtained using all cores during the Stream execu-

tion. We can observe that the Intel4x8 machine presents the best memory bandwidth due to its QPI network technology. The last level cache bandwidth, for each machine is also presented in the table. To compute this performance metric, we have used the sequential version of Stream benchmark on one node of each machine. In order to obtain the cache memory bandwidth, the problem size used as input for Stream is smaller than the last level cache. NUMA factors are shown in intervals, meaning the minimum and maximum penalties to access a remote DRAM in comparison to a local DRAM.

Table 7.1: Overview of the Multi-core Platforms.

Characteristic	**AMD8x2**	**Intel4x24**	**Intel4x8**
Number of cores	16	96	32
Number of processors	8	16	4
NUMA nodes	8	4	4
Main Memory bandwidth (GB/s)	9.77	4.1	35.54
Cache bandwidth (GB/s)	8.69	6.41	15.12
Cache-Coherence	MOESI	MESI	MESIF
Interconnection	HyperTransport	FSB	QuickPath
NUMA factor (min/max)	[1.1; 1.5]	[2.2; 2.6]	[1.36; 3.6]

7.2 Software Stack

Considering the software stack used in our experiments, we have divided into two groups, the basic software and the statistical/analysis softwares. The basic software is composed by the operating system and compilers used in the machines. The statistical/analysis software allows us to collect information during an execution to better understand the behavior of the environment.

All machines run GNU/Linux operating system, kernel version 2.6.32 with NUMA support, NUMA API [Kleen 2005]. As compilers we use GNU C Compiler (GCC) and Intel C Compiler (ICC). On both AMD8x2 and Intel4x8 the compiler versions are 4.4.4 for GCC and 11.0 for ICC. The Intel4x24 uses the versions 4.4.5 for GCC and 11.1 for ICC.

Since we are interested in the impact of data placement on the applications performance, we have selected some user space tools that provide information of the memory subsystem of NUMA machines. Considering memory management, we select *vmstat*, *numactl* and *numastat* tools to observe memory behavior during the application execution. These tools provide information of memory utilization on the machine. Furthermore, *numastat* allow us to observe the NUMA hit/miss for allocations performed in the platform.

In order to get information of performance hardware counters from such machines, we use some software support such as Performance Application Program-

ming Interface - PAPI [PAPI 2010], Intel Performance Analyzer Tool - vTUNE [Intel-vtune 2010] and Intel Performance Tuning Utility - PTU [Intel-PTU 2010]. PAPI is a performance interface that extracts processor events and it supports several processors. It provides a high level interface that simplifies the work of extracting performance hardware counters for an application. vTUNE and PTU are performance tools developed at Intel that reports processor events. Both tools work with the sampling method, which uses just a subset of the events observation to generate the information for users. vTUNE is graphical tool that presents performance graphs whereas PTU is a text based tool that generates large tables for presenting the events. vTUNE and PTU are complementary tools for performance analysis.

Performance hardware counters provide more precise information about the system behavior, allowing us to have a better overview of the relation between application and processor events. Some of the information that PAPI, vTUNE and PTU tools can provide are cache hit/miss, DRAM local/remote access, interconnection utilization, processing units usage and number of instructions of an application. Considering PAPI, we have used it to get the total number of processor cycles consumed for an application and the total number of instructions of the considered application. vTune and PTU have been used in the Intel machines to retrieve information of cache memories access and DRAM accesses. These information provide an overview of how a selected data and thread placement strategy impacts in the overall performance of an application.

7.3 Performance Metrics

In order to evaluate the performance of Minas, it is important to use metrics that allow a better understanding of the memory performance of applications on the multi-core NUMA machines. Therefore, our performance evaluation is based on metrics and hardware counters that are related with the memory sub-system. We select as metrics: memory usage, speedup, execution time and latency to access data. Considering the performance hardware counters, the selected ones are CPU cycles, cache miss, local/remote cache access and local/remote DRAM access.

The memory usage metric allows us to better comprehend application memory consumption, how application data is physically allocated and placed over the physical memory banks of the machine. Therefore, this metric is split in three other metrics: memory consumption; allocation hit/miss and physical memory bank usage. To retrieve these values we use *vmstat*, *numactl* and *numastat* tools. Memory consumption comprises virtual and physical memory. Data allocation hit/miss is related with the NUMA node capacity. For instance, if any data allocation can not be performed locally it will generate an allocation miss that is retrieved with *numastat* tool. The physical memory banks usage allows us to verify if data placement is balanced over the machine nodes.

For all applications and benchmarks, the execution time is the time spent to execute the role application/benchmark. Therefore, the execution time includes the

time to perform memory accesses and computation. The speedup is then the ratio between the execution time of the sequential (Tseq) and the parallel (Tpar) versions of the considered application/benchmark. Therefore, the speedup for an application run in n cores is defined by:

$$S(n) = \frac{Tseq}{Tpar} \tag{7.1}$$

Finally, the selected performance hardware counters used in our analysis are obtained using PAPI, PTU and vTUNE tools. CPU cycles has been obtained with PAPI interface. The cache miss, local/remote cache access and local/remote DRAM access events have been retrieved using vTUNE and PTU.

7.4 Measurement Methodology

We execute each experiment multiple times (minimum of 30 executions), obtaining small standard deviations (up to 4%). We use the arithmetic average to compute the final values of each metric.

Considering the problem sizes, we work with sizes that fit and do not fit on the last level cache of each machine. Such approach allows us to better analyze the impact of memory affinity management on the different levels of the memory sub-system of the NUMA platform.

Chapter 8

Evaluation on OpenMP Benchmarks and Geophysics Applications

In this chapter, we present the performance evaluation of Minas framework. Our performance evaluation relies on experiments with synthetic and numerical scientific benchmarks and two real geophysics applications on three NUMA platforms. We consider benchmarks and real applications to explore the different application memory access characteristics (e.g. memory access mode, access patterns) that have an important impact on the memory affinity management. Our aim is to analyze the impact of different architecture characteristics such as bandwidth and NUMA factor in the memory affinity management for parallel applications. We compare Minas results to the ones obtained with the standard memory affinity management on Linux operating system, with the NUMA API, the numactl tool and the thread affinity interface of GNU C Compiler/Intel C compiler. We first describe each one of the benchmarks/applications. After that, we present their results and analysis. In this chapter, we focus in the C implementation of all selected benchmarks/applications with OpenMP for code parallelization.

8.1 Synthetic Experiments

In this section, we present Minas MAi evaluation with a synthetic benchmark to better understand its performance. We start with the performance evaluation of MAi because it is the core of Minas framework. Using the synthetic benchmark, we evaluate MAi memory allocators and memory policies overall performance.

The synthetic benchmark implements a regular and an irregular computation on two 2-dimensional matrices. The regular computation is a jacobi operation whereas the second one uses indirect indexes to access the two-dimensional matrices. In the jacobi operation to compute a new value for an element on the two-dimensional matrix, it is necessary to retrieve the current values of its four neighbors (i.e. north, south, east, west). Based on these values, an average of all four values is computed and attributed to the element. In the case of the irregular computation, the two-dimensional matrices are accessed in a random way, which reduces the cache usage for the operation. These characteristics enable us to evaluate applications that demands both shorter latency and high bandwidth for memory accesses.

The benchmark has two steps: the data allocation/initialization and the computation. In the first step, data is allocated by the master thread and initialized by the worker threads. After that, the computation step performs the regular and the irregular computations. Figure 8.1 shows a snipped of the benchmark code. The first two OpenMP loops perform the regular jacobi operation, whereas the other two perform the irregular jacobi operation.

```
//Regular computation - jacobi operation
for (iters = 1; iters <= numIters; iters++) {

  #pragma omp parallel for private(j)
    for (i = 1; i < gridSize; i++)
      for (j = 1; j <= gridSize; j++)
        grid2[i][j] = (grid1[i-1][j] + grid1[i+1][j] +
                       grid1[i][j-1] + grid1[i][j+1]) * 0.25;

  #pragma omp parallel for private(j)
    for (i = 1; i < gridSize; i++)
      for (j = 1; j <= gridSize; j++)
        grid1[i][j] = (grid2[i-1][j] + grid2[i+1][j] +
               grid2[i][j-1] + grid2[i][j+1]) * 0.25;
  }

//Irregular computation - random accesses
  for (iters = 1; iters <= 2*numIters; iters++) {
    for (i = 1; i < gridSize; i++) {
    #pragma omp parallel for
     for (j = 1; j <= gridSize; j++)
        grid1[i][j] = (grid2[i-1][rand_n[j]] +
        grid2[i+1][rand_n[j]] + grid2[i][j-1] + grid2[i][j+1]);
      }

   for (i = 1; i < gridSize; i++) {
     #pragma omp parallel for
     for (j = 1; j <= gridSize; j++)
        grid2[i][j] = (grid1[i-1][rand_n[j]] +
         grid1[rand_n[i]][j] + grid1[i][rand_n[j]]
            + grid1[rand_n[i]][j+1]); }
  }
}
```

Figure 8.1: The Synthetic Benchmark Computation Kernels.

For the experiments with this benchmark, we have used all cores of the three machines described in section 7.1, a problem size of 8192x8192 and 100 iterations. Since we aim at evaluating the memory performance of our solution, we selected this size for the matrices to reduce cache influences in our memory accesses. We compare the Minas MAi results to the ones obtained with the default memory affinity strategy of Linux operating system the *first-touch* and to the ones obtained

with the tool *numactl*. Considering Minas and *numactl*, threads have been pinned to the cores. In the case of *first-touch*, we let Linux kernel schedule all threads and perform any necessary thread migration.

Table 8.1 shows the execution time for the regular computation of the benchmark when executed with one thread per core on the selected machines. The presented execution time considers the time to allocate, to initialize and to perform the jacobi operation. Since the computation is very regular (four-point stencil), we have used the *mai_bind_block()* to place data on the machine memory banks. This memory policy places data closer to the threads that work on them. Therefore, the policies *first-touch* and *mai_bind_block()* present the best performances. Because of the similarity on the behavior of these two memory policies, results obtained with MAi, *first-touch* and *numactl* do not present any significant difference on all the machines. However, *first-touch* and *numactl* have applied the memory policy on all application data whereas on Minas MAi, we have applied the memory policies only on the arrays used by the jacobi computation. Therefore, we can conclude that control memory affinity for all application data is not necessary. Memory affinity can be applied only on data that generate more NUMA penalties for the application.

Table 8.1: Execution Time in seconds (s) for Benchmark

Machine	**Minas MAi**	**First-Touch**	**Numactl**
AMD8x2	26.37	26.26	27.53
Intel4x24	40.95	39.42	40.02
Intel4x8	7.13	8.52	8.54

Particularly, we can observe that Minas MAi has obtained better results for the Intel4x8 machine. This is mainly because of the strategy that Minas MAi uses to allocate arrays, in which it allocate arrays on separates heaps, avoiding the usage of the same memory block by different arrays. Besides the allocation, Minas MAi avoids thread migration, favoring cache memory usage. Additionally, this machine has the highest NUMA factor compared to the other two machines. This means that manage memory affinity efficiently generates better performances for the application.

In Table 8.2, we present the execution time for the irregular computation of the benchmark when executed with one thread per core on the selected machines. Since the benchmark has two distinct phases (initialization of the arrays and the irregular computation), in the case of Minas MAi we used different memory policies for each phase. We start with the *mai_bind_block()* policy for the initialization phase and then, we changed for the *mai_cyclic_neighbors()* to increase memory bandwidth for threads to access data in the irregular phase. In this case, some memory pages migration are performed by the Minas framework between the two phases to correct data placement. However, between the two phases threads mapping in the machines cores were not changed.

Considering *first-touch* and *numactl*, we used the same memory policy for the

all execution, because on both cases there is no support for multiple memory policies in the same execution. For the *numactl* tool, we applied the interleave memory policy and we pinned each thread to a core of the machine. We use the interleave memory policy because it is the closest one to the MAi memory policy *mai_cyclic_neighbors()*. We can observe that Minas MAi has obtained better performances for the two Intel machines but not for the AMD8x2 one. Since remote accesses on the AMD8x2 machine are not expensive (small NUMA factor), the costs to perform data migration for the second phase of the benchmark are generally higher than the cost of remote accesses. Due to this, Minas MAi has obtained worser results than *first-touch* and *numactl*. However, in the case of the two Intel machines, data migration has improved the overall performance of the benchmark up to 17%.

Table 8.2: Execution Time in seconds (s) for the Synthetic Benchmark

Machine	Minas MAi	First-Touch	Numactl
AMD8x2	646.195	596.115	596.368
Intel4x24	1024.65	1134.664	1090.744
Intel4x8	212.847	237.611	257.528

8.2 Experiments with Benchmarks

In this section, we present the performance evaluation of Minas with Stream Benchmark [Mccalpin 1995] and NAS Parallel Benchmarks [Jin 1999]. The selected benchmarks allow us to evaluate Minas data allocation, data and thread placement for applications with different characteristics (memory consumption and regular/irregular data access) and requirements (memory bandwidth and latency).

8.2.1 Stream Benchmark

Stream is a micro benchmark which is largely used to evaluate memory bandwidth performance of parallel machines [Mccalpin 1995]. It is a synthetic benchmark application that measures the aggregated memory bandwidth for different memory access patterns. To compute such a metric, Stream uses three vectors and four operations (copy, scale, add, triad). Additionally, in order to reduce the cache influence on the results, the problem size is larger than the size of the last level cache.

Table 8.3 shows Stream operations and their specification. As we can observe, all operations are performed with double vectors. The copy operation allows the user to measure transfer rates between processing unit and memory bank. The operation scale adds a multiplication by a scalar to the copy operation. Sum allows the users to verify memory system performance when multiple loads/stores are performed.

The operation triad is a merge of all operations (copy, scale and sum). All of these operations are computed in separated parallel loops, one for each operation.

Table 8.3: Stream operations

Operation Name	Operation	Data type
Copy	a[i] = b[i]	double
Scale	a[i] = q*b[i]	double
Sum	a[i] = c[i]+b[i]	double
Triad	a[i] = c[i]+q*b[i]	double

We use three versions of Stream, the original one without modifications and two tuned versions with random/irregular access to the vectors. In original version, each thread computes a chunk of the vectors (regular access) that are scheduled in a static way. This means that the chunk size is equal for all threads, except for the last thread that can have a larger chunk size (if the number of elements of vectors are not divisible by the number of threads). In the tuned version with random access, the chunk for each thread is also scheduled in a static way, but random indexes to access vectors are used. The second tuned version uses a dynamic scheduling of work in the computation phases. Figure 8.2 shows the main difference of the two versions of Stream Benchmark for add operation.

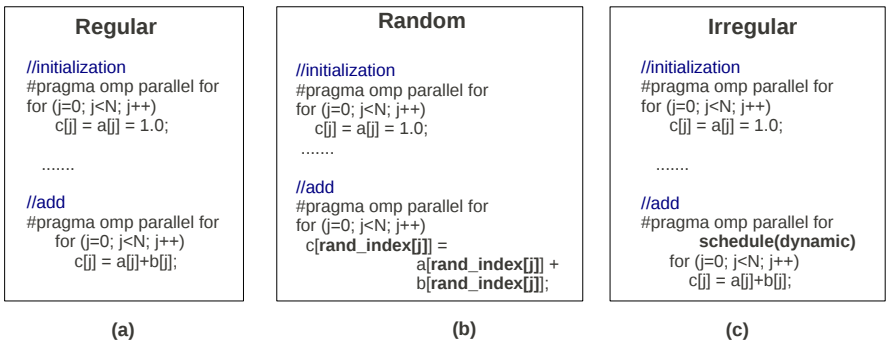


Figure 8.2: Stream Benchmark: (a) Original Version (b) Tuned Version with Random Access (c) Tuned Version with Irregular Access.

Since Stream has been implemented with static arrays, we use it to evaluate the performance of Minas MApp using the GCC and ICC compiler. For the experiments, we have used all cores of the Intel4x8 machine described in section7.1, a problem size of 4Gbytes and 10 iterations. We present the results only for the Intel4x8 machine because it allows us to evaluate MApp with GCC and ICC compilers. Additionally, similar results have been obtained for the other two machines. The chosen memory affinity mechanisms to compare with Minas MApp performances

are the affinity interfaces of GCC [1] and ICC [2]. These compilers support affinity by using some environment variables (see section 3.5.1.1) to select a thread mapping strategy.

Table 8.4 reports the memory bandwidth for Stream benchmark considering the three versions of the code. The results shown here represent the aggregated memory bandwidth for the triad operation with the memory affinity solution of MApp, GCC and ICC. The other operations are not presented in the table because they have presented similar results. Additionally, the triad operation is a merge of all other operations. The three memory affinity solutions present similar results. On general, the compilers memory affinity mechanisms have obtained better results for the regular version of Stream. This happens because in the regular version of this benchmark, threads do not share any data. Therefore, the compilers improve memory affinity because it places memory pages closer to the threads that use them. They use the first access to memory pages to perform data placement, which is actually performed by the operating system. Compilers memory affinity support only maps thread to cores to avoid any thread migration.

In contrast, for the random and irregular versions of Stream, MApp has presented some slightly performance improvements compared to the other solutions for some cases. In the random version, threads share data and place data closer to them considering only the first access does not guarantee memory affinity. Due to this, MApp select to spread data over all the machine nodes in order to enhance memory bandwidth usage. Considering the irregular case, MApp also spread data over the machine memory banks. However, in the GCC compiler this strategy does not improve memory affinity. After analyzing the benchmark execution, we observe that its work distribution for threads has been similar to the regular one. As a consequence, MApp does not improve the benchmark performance. An opposite behavior has been observe for ICC, which make MApp strategy be profitable for the benchmark.

Table 8.4: Memory Bandwidth (MB/s) for Stream Triad Operation

Version	Minas MApp - GCC	GCC Affinity	Minas MApp - ICC	ICC Affinity
Regular	32000.6	35578.7	31733.1	35879.4
Random	3428.29	3394.54	3438.06	3516.3
Irregular	441.474	594.969	988.33	950.93

Although the compiler affinity strategies have provided some performance improvements for the Stream benchmark, it is important to emphasize that they are transparent solution in the sense of no source code modifications. However, the user

1. GCC Thread affinity interface - http://gcc.gnu.org/onlinedocs/libgomp/Environment-Variables.html
2. ICC Thread affinity interface - http://software.intel.com/en-us/intel-compilers/

has to explicitly control affinity choosing which nodes and cores must be used for the application execution. Contrary to this, the MApp solution is automatic, the user does not have to interfere in the application optimization. Additionally, it is also important to mention that this benchmark represents an extreme case, where there is several memory accesses and small computations. Although this benchmark is memory bound, its execution time is small, which reduces the benefits of memory affinity management. In order to evaluate the performance of Minas with more realistic applications, the next section presents results for the NAS Parallel Benchmarks.

8.2.2 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) is a benchmark derived from computational fluid dynamics (CFD) codes. It is composed by a set of applications and kernels [Jin 1999]. NPB is a representative HPC benchmark of numerical computation and data communication. Additionally, they are examples of memory-bound and CPU-bound programs. These characteristics allow us to better investigate the impact of memory affinity management on multi-threaded programs over multi-core machines with NUMA design. NPB has been implemented in a number of languages, using different strategies and algorithms for code parallelization. In this work, we use the OMNI compiler group C implementation of NPB version 2.3 [Omni Project 2010].

From NPB version 2.3, we selected seven kernels/applications: Fast Fourier Transform (FFT), Multigrid (MG), Lower and Upper triangular system solution (LU), Conjugate Gradient method (CG), Block Tridiagonal equations solution (BT), Solution of Pentadiagonal equations (SP) and Embarrassingly Parallel (EP). These kernels/applications were chosen due to their memory access patterns (regular and irregular data access) and different data structures types. Additionally, they represent important classes of algorithms and computations of HPC applications. The IS (Integer Sort) and UA (Unstructured Adaptive) applications, which are also part of the NPB benchmark, were not used in this work. UA was not implemented in this version whereas IS presented some problems during the execution. Table 8.5 summarizes the description and characteristics of the selected benchmarks.

For the experiments with these benchmarks, we use three classes of problem size: A (small), B (medium) and C (large). Table 8.6 reports the memory consumption for the three classes on the selected benchmarks. We can observe that some of the benchmarks (e.g. EP and MG) have the same memory consumption for different classes. In MG benchmark, the difference between classes A and B are only in the number of iterations and not in the size of data structures. The varying problem sizes allow us to investigate and comprehend the performance of Minas MAi mechanisms when different amount of memory are used by the application.

Table 8.5: Selected Applications from NPB.

Name	Description	Structure, computation and communication patterns
FFT	Computes the fast Fourier Transform for three dimensional systems.	3D matrix, one dimension computed in parallel at a time, long-distance communication.
MG	Uses a V cycle MultiGrid method to calculate the solution of the scalar Poisson equation.	3D dense matrix, 3D grid using 27-point stencils, short-distance communication.
LU	Solves a 3D seven-block-diagonal system using LU triangular systems solution.	Dense matrix, work decomposition, both continuous and non-continuous communication.
CG	Uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix.	Large sparse matrix, sparse matrix-vector multiplication, long-distance communication.
BT	Computes a solution for multiple and independent systems of non-diagonally dominant.	Dense matrix, block tridiagonal solvers, non-contiguous communication.
EP	Implements a parallel random-number generator.	No special data structure, upper bound float-point performance, no communication.
SP	Computes the solution for a scalar pentadiagonal systems.	Dense matrix, block tridiagonal solvers, non-contiguous communication.

Table 8.6: NPB Problem Sizes in MBytes for Each Class.

Class	FFT	MG	LU	CG	BT	EP	SP
A	427.22	439.00	45.64	56.16	299.69	33.00	79.69
B	1697.06	439.00	174.51	400.20	1201.94	33.11	315.00
C	6724.08	3432.88	677.32	1039.65	4793.77	33.11	1249.13

8.2.2.1 Overall Performance

In this section we present the experimental results for the NPB with Minas MAi and Linux *first-touch* policy on two of the machines presented in section 7.1, the AMD8x2 and the Intel4x8. We do not present results for the Intel4x24 because the NPB does not scale with so many cores. In our analysis, we have used the execution time as the measurement metric to evaluate the performance of Minas MAi. It is important to mention that we have used all cores of the machines with one thread

per core. Additionally, we have not done any modifications to control threads and data locality on the Linux results.

For thread placement on Minas, we have used memory traces of the benchmarks in order to have a thread placement that take into account data sharing. The memory traces were used to identify which threads access the same shared memory range. Using this information, we model the data sharing between threads as a complete graph. To do so, the vertices represent the threads and the edges the amount of data sharing among threads. This graph is then processed by a matching algorithm [Cruz 2010], which gives as result the groups of threads so that the amount of data sharing is maximized. Based on this information, we computed thread placement and provided to Minas an input configuration file with the mapping between threads and cores. The work [Cruz 2011] presents more details of our strategy to compute thread placement. The maximum standard deviation for the results presented in this section is 3.5%.

Figure 8.3 shows the execution time obtained with the NPB benchmarks for the three problem sizes on the AMD8x2 machine. One can notice significant performance gains for CG, FFT and MG benchmarks on this machine when using Minas MAi mechanism. Considering EP, BT and SP benchmarks, Minas had similar performances compared to the Linux whereas for LU, Linux had better performances. Particularly, this machine does not have any shared cache memory, data sharing between threads is related only to global memories. Therefore, Minas MAi memory policies has an important role on the improvement of memory affinity, improving latencies and bandwidths to get data.

On the AMD8x2 machine, the performance gains with CG, FFT and MG benchmarks relative to the Linux operating system results were 35%, 35% and 55% on average. These benchmarks have different memory access on different shared arrays. Consequently, placing data considering such differences reduces the number of remote access to get data and increases the available memory bandwidth. Furthermore, in the AMD8x2 machine, bandwidth is an important issue. Due to this, it is important to guarantee load balancing and less memory contention using the global memories available on the machine. For these benchmarks, we use cyclic memory policies (e.g.*mai_cyclic()*, *mai_cyclic_neighbors()*) for arrays with a high level of sharing in order to ensure good bandwidth for threads. For arrays with a low level of sharing and regular accesses we use the bind memory policies (e.g. *mai_bind_rows()*).

For the BT, SP and EP benchmarks, Minas MAi has not presented any significant improvement gains compared to the Linux mechanism. Considering BT and SP benchmarks, we have observed that some threads do not share data. Additionally, most of the parallel sections have the same data access as the initialization step. This behavior is favorable to Linux, since it uses first accesses on data to place it on the machine. Due to this, Minas MAi does not have a significant impact on their performance.

In the case of EP, this benchmark is CPU-bound and threads perform independent computation on their private data. Thus, any improvement gains can be

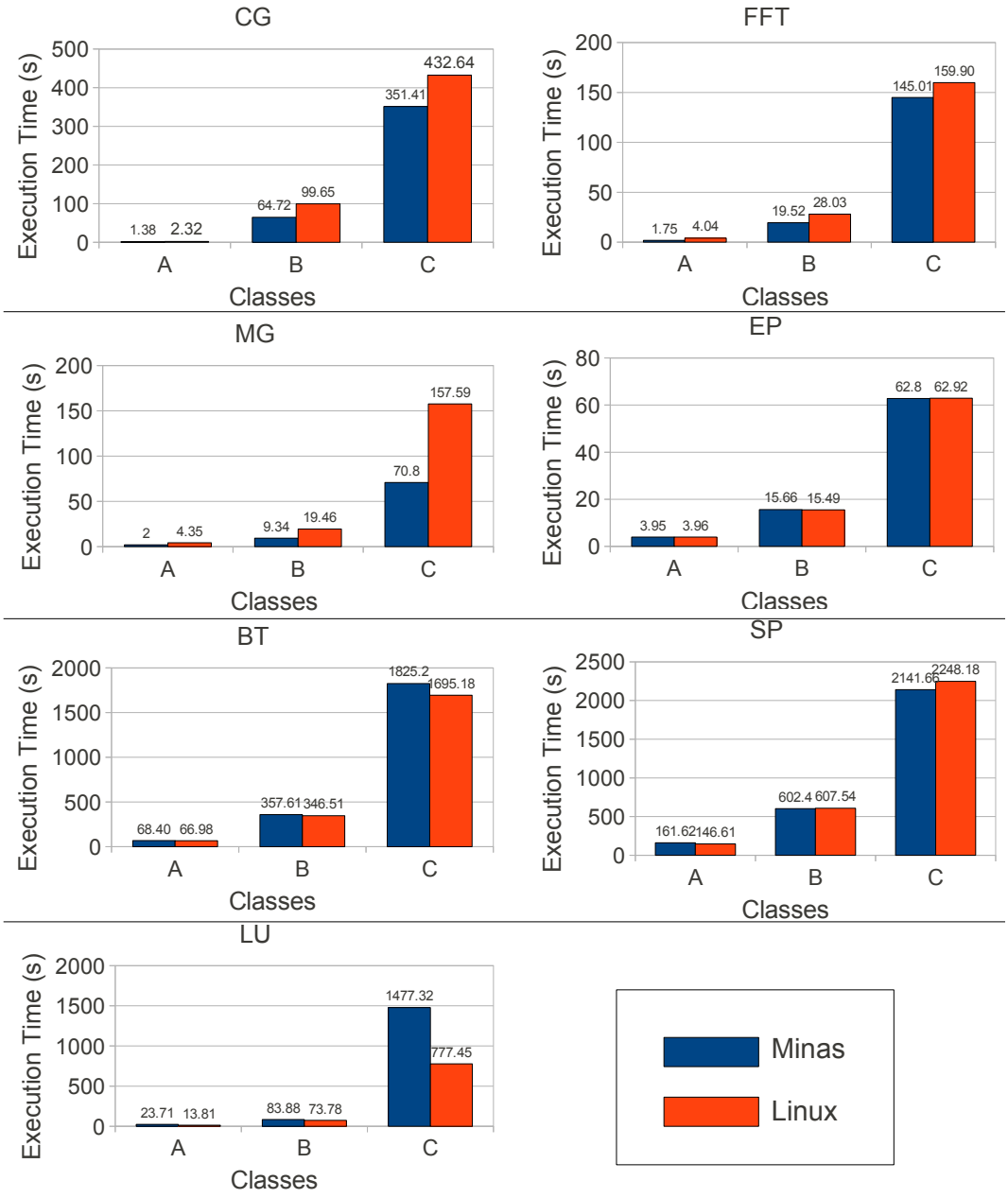


Figure 8.3: NAS Parallel Benchmarks on AMD8x2.

achieved for this benchmark when using our method. The LU benchmark has a high level of heterogeneity on data accesses, regular accesses interleaved with irregular accesses. Because of this, we have to change memory policies when using Minas MAi through the different steps of the application. In order to change the memory policy associated with some data, Minas MAi performs data migration that has generated important overheads for this benchmark. Due to these overheads, the

results obtained with Minas are worse than the ones obtained with Linux.

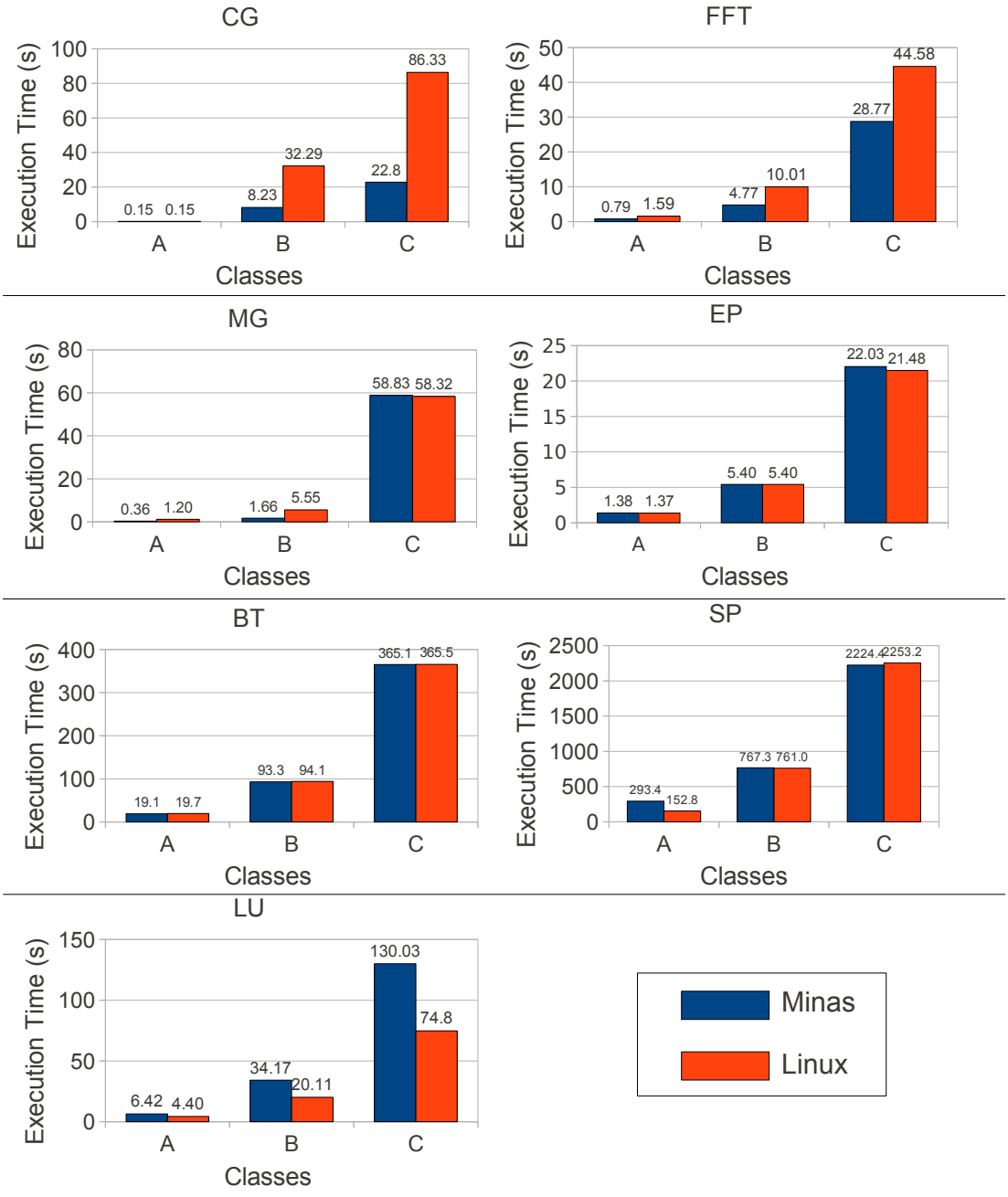


Figure 8.4: NAS Parallel Benchmarks on Intel4x8.

Figure 8.4 reports the execution time obtained with the benchmarks on the Intel4x8 Machine. Similar to the results obtained with the AMD8x2 machine, we observe some important performance improvements for CG, FFT and MG benchmarks. For the BT, SP and EP benchmark Minas had similar performance compared

to Linux whereas for LU, Linux had better performance.

Considering CG, FFT and MG benchmarks gains are up to 75%, 50% and 70% (except for class C) respectively when compared to the Linux standard solution for memory affinity. Since these benchmarks are more sensitive to memory access and considering the characteristics of the machine, place data correctly reduces the NUMA impact on the application performance. Additionally, in the case of Linux, the *first-touch* memory policy does not garantee a balanced usage of the machine memory banks in the data placement. Minas MAi supports different memory policies within a same application, which reduces the NUMA penalties such as load balancing, memory contention and remote access. In the case of CG, FFT and MG, we guarantee load balancing and less memory contention using all the DRAM memories available on the machine whereas, the operating system has placed more data on some restrict DRAM memories.

In the CG and MG benchmarks, the main characteristic is the indirect access by threads on some arrays. Due to this, it is more difficult to Linux mechanism to perform an efficient thread and data mapping for them. In the case of scheduling, Linux does not take into account threads access to memory hierarchy. For these experiments, we have observed that Linux generated more cache misses and remote accesses, re-scheduling threads to different cores. Considering data placement, Linux uses the *first-touch* memory policy. Thus, only the first access by threads on data are considered for data placement on the DRAM memories. Contrary to this strategy, Minas MAi places threads over the machine cores considering its cache memory hierarchy. Furthermore, Minas MAi supports data migration, allowing developers to change data placement over the different steps of the applications.

Minas MAi has not improved the performance for BT and SP because of their regularity on data access on the parallel sections. In this case, both Minas MAi and Linux use the same memory policy to enhance memory affinity. Considering EP benchmark, in the case of Linux, we have observed that it keeps threads on the same cores during its execution. Since this benchmark is CPU-bound, both Minas MAi and Linux use the same affinity strategy and consequently, they have obtained similar results. As mention on previous paragraphs, LU has regular accesses on data interleaved with irregular ones. Because of this, we have to change memory policies when using Minas MAi through the different steps of the applications. In order to change the memory policy associated with some array, Minas MAi performs data migration that has generated significant overheads that were not amortized in the execution time of the benchmark.

8.2.2.2 Understanding Performances

In order to have an insight about the Minas memory affinity impact, we have selected two benchmarks that have been or not impacted by Minas, *i.e.*, EP and MG (both using Class B). These benchmarks are very distinct: EP is CPU-bound while MG is memory-bound. Considering these benchmarks, we can use the information of how threads access data and how they share data to investigate the impact of the

Minas memory affinity management. Therefore, to better understand data sharing of EP and MG benchmarks, we have run both benchmarks on Simics Simulator [VIRTUTECH 2007] with a selected memory access tracing tool.

We have used the tool inside Simics proposed in [Cruz 2010] to trace the memory accesses of the benchmarks and then generate what authors named shared matrix. The shared matrix allow us to better understand EP and MG data sharing, since it provides the communication pattern between threads. For this simulation, we have used 16 threads and the simulated machine parameters are similar to the AMD8x2 platform. After that, we have used the vTune and PTU tools to obtain some performance hardware counters while executing the benchmarks on the Intel4x8 machine. We use AMD8x2 platform for the memory access patterns because the other two machines have a large number of cores, which complicates the visualization of memory access patterns for all threads. The Intel4x8 machine is used for the performance hardware counters because it is the only one with vtune and PTU softwares.

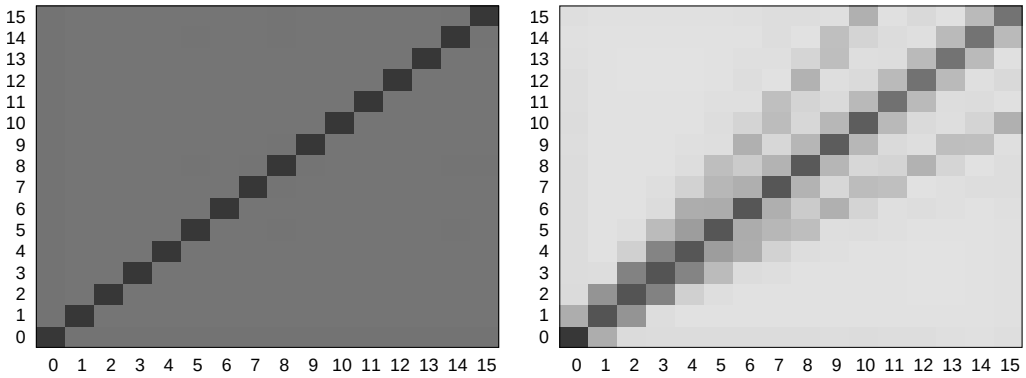


Figure 8.5: Shared Matrix for EP (a) and MG (b) Benchmarks.

Figure 8.5 reports the shared matrix generated by the tool proposed in [Cruz 2010]. In order to generate this matrix, the tool reads the memory traces generated by Simics and evaluates how much shared memory each thread uses and how many access were performed by each thread in the shared data. The shared matrix represents the number of access performed to a memory block that is shared by some threads. In the figure 8.5, each cell (i, j) represents the data sharing between threads i and j. When i equals j, it represents the accesses to the private data. A darker cell means high level of data sharing between a pair of threads.

In the figure, we can observe that EP has homogeneous sharing pattern, hence it is not impacted by thread and data placement. Contrary to this, in the MG benchmark that the thread 0 probably does some initialization or post-checking of the data, since it access most of the shared memory (cell(0,0) is darker than other ones). In the Minas version of MG, we have optimized this behavior by placing shared data among the NUMA nodes. Due to this, the improvement gains of MG on both machines have been significant. The MG benchmark has a communication

pattern in which the nearby threads share more data. Since Minas default thread placement enhance cache sharing, MG communication pattern is benefited by this placement.

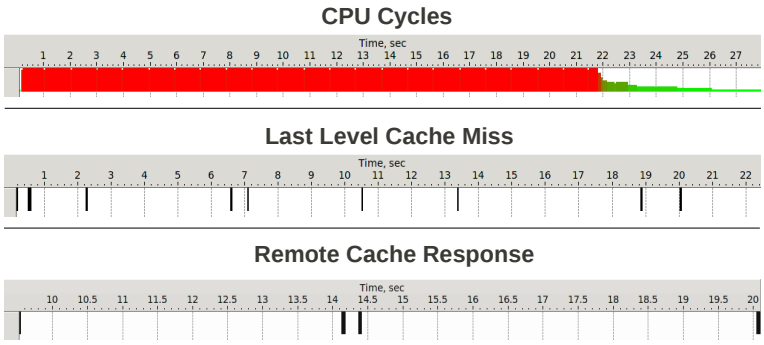


Figure 8.6: Event Counters of EP on Intel4x8.

Figure 8.6 shows the CPU cycles, last level cache miss and remote cache response performance event counters for EP, whereas the Figure 8.7 shows the same event counters for MG on the Intel4x8. In this figure, the black color means few events, green some events and read several events. Since this machine has NUMA characteristics, it is important to investigate the ratio between remote accesses and CPU cycles on both benchmarks, to comprehend the importance of data placement for the application. On this machine we have used vTune to extract information of accesses on local and remote last level caches. We can observe that the main difference between EP and MG is the number of cache accesses during the benchmark execution. EP has presented almost no access to the last level cache, while MG performs several accesses on the last level cache memory. Such results let us to conclude that MG is much more sensitive to memory placement than EP. Due to this, the performance improvements in MG are higher than in EP.

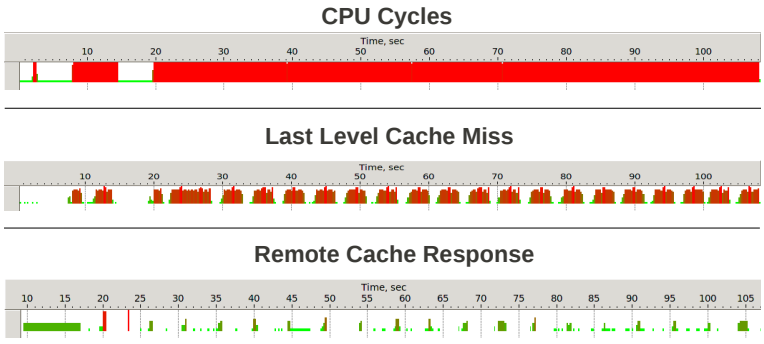


Figure 8.7: Event Counters of MG on Intel4x8.

We also investigate the impact of Minas memory affinity management for both benchmarks using PTU. This tool provides several hardware counters that allow us to better understand the performance presented on the previous section for EP and MG. The selected performance hardware counters are total CPU cycles, cache L3 miss (L3_misses), local and remote caches (L_cache and Rem_cache) and, local and remote memory banks (DRAM) access (L_DRAM and Rem_DRAM).

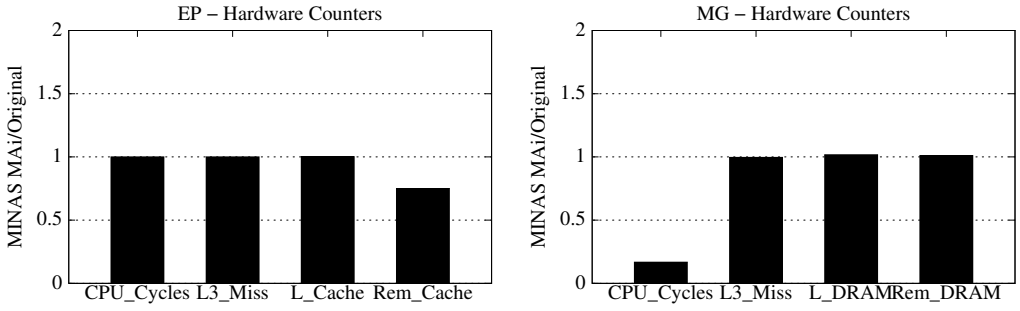


Figure 8.8: Event Counters on Intel4x8 for EP (a) and MG (b).

Figure 8.8 shows performance events counters for EP and MG benchmarks on Intel4x8 Machine, compared to the original execution with Linux memory affinity mechanisms. In Figure 8.8 (a), we can observe cache miss and access rates on the Intel4x8 machine for EP benchmark. We noticed that there are slightly differences between Minas MAi to the original execution. However, they do not impact on the overall performance of EP benchmark. Execution times for the two versions are similar, with small differences in the number of instruction executed per second. Considering the cache L3, the Minas MAi strategy and Linux have generated similar data misses. Such results let us to conclude that on applications with characteristics like EP (independence between threads and no shared memory) do not suffer with the NUMA design.

Figure 8.8 (b) shows that Minas MAi has presented similar number of cache L3 misses and accesses to DRAM to the Linux (original). However, it can be noticed that Minas MAi strategy have expressively reduced the total number of CPU cycles. Since it controlled data placement for each step of the application over the machine memory banks, it increases memory bandwidth and reduces latency for all cores to get data. In MG benchmark, computation is performed by zones with irregular accesses in these zones. By controlling data over all available memory banks, Minas MAi allows much more memory pages of a zone to be accessed by threads in the same interval time.

8.3 Geophysics Applications

In this section, we present performance evaluation of Minas on two real applications from Geophysics. These applications allows scientists to better understand the geographic characteristics and events of an region. First, we introduce the Ondes 3D application that performs simulation of Seismic Wave Propagation [Dupros 2008, Dupros 2009, Ribeiro 2010c]. After that, we present the Interval Categorizer Tessellation Model (ICTM) [Castro 2009b, Ribeiro 2009a] that classifies a geographic region considering its characteristics. Ondes 3D and ICTM represent important memory-bound numerical scientific problems that demand both low latency and high memory bandwidth for memory accesses.

8.3.1 Ondes 3D: Simulation of Seismic Wave Propagation

Ondes 3D is a application that simulates seismic wave propagation in three dimensional geological media based on finite-difference discretization [Dupros 2008, Dupros 2009]. It has been developed by the French Geological Survey (BRGM - www.brgm.fr) and it is mainly used for strong motion analysis and seismic risk assessment. The particularity of this application is to consider a finite computing domain even though the physical domain is unbounded. Therefore, the user must define special numerical boundary conditions in order to absorb the outgoing energy.

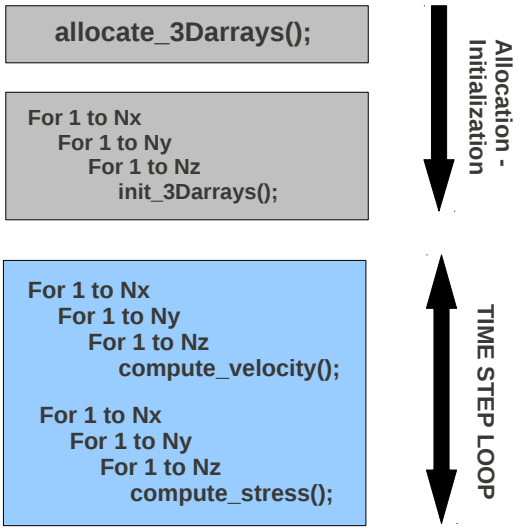


Figure 8.9: Ondes 3D Application.

Ondes 3D has three main steps: data allocation, data initialization and propagation computation (composed by two computation loops). During the first two steps, the three dimensional arrays that represents a simulation are dynamically allocated and initialized. These two steps are very important because data are touched and

physically allocated in the memory banks of the machine. During the last step, the two computation loops compute velocity and stress of the seismic wave propagation. In all steps, the three dimensional arrays are accessed on write only, read only an write/read mode. Another important characteristic of this version of Ondes 3D is that it has only regular memory access patterns on data. By regular access, we mean that threads always access the same elements of the arrays in the same order. Moreover, in Ondes 3D, memory bandwidth to get data is also important, since in some of its steps threads have a high level of data sharing. Figure 8.9 presents a schema of the application with its three steps. Ondes 3D has only short distance memory accesses, only few neighbors of each array element are needed for computation.

We carry out experiments with problem size of 2.6 Gbytes (larger than cache memories) and we use one thread per core of the machine. We compare Minas MAi mechanism with some solutions for memory affinity on Linux, the *first-touch*, the *numactl* and the *libnuma*. Since the original version of Ondes 3D relies on dynamic memory allocation, we do not consider Minas MApp on the experiments with it. Experiments with different problem sizes and other NUMA machines are presented in [Dupros 2009, Ribeiro 2010b, Dupros 2011].

Regarding the Linux memory affinity solutions, we have changed the application source code or their executions parameters. In order to use *first-touch* and *libnuma*, we have changed data allocation and initialization. In the case of *first-touch*, we have two versions, one named master initialization and other one named thread initialization. In the master initialization (original version of the code), only the master thread initializes all the arrays whereas in the thread initialization each thread initializes its own data (our modification). Considering the *libnuma* API, we have allocated data with the *numa_ alloc()* and *numa_ alloc_ interleaved()* functions. For the *numactl*, we run the thread initialization version of the application with the option *physcpubind* to pin threads to cores to avoid any thread scheduling by Linux. In this way, we ensure that threads do not lose the applied memory affinity strategy. We have also used the option *interleave* for the *numactl*, in order to provide good bandwidth for the threads.

The Minas MAi version of the code has been implemented by applying the most suited memory policy for each array of the application. Depending on the array access and platform, we have chosen one of the following memory policies (*cyclic, skew map* and *bind block*). The first two memory policies are ideal for shared arrays that have their elements accessed by different threads in read only and read/write mode. Since *cyclic* and *skew map* spreads data among nodes, they improve bandwidth for data accesses. The latter memory policy is suitable for arrays accessed in a regular way, where threads always access the same data set in write only mode. The reason to use *bind block* for write only arrays is to avoid penalties that come from the cache coherence protocol of NUMA machines. We first present the results for Minas MAi and *first-touch* on the three machines and then, the results comparing Minas MAi, *libnuma* and *numactl* on the AMD8x2.

Figure 8.10 reports the execution time for Ondes 3D (problem size of 2.6Gbytes and five iterations) when executed on the three multi-core NUMA machines. We

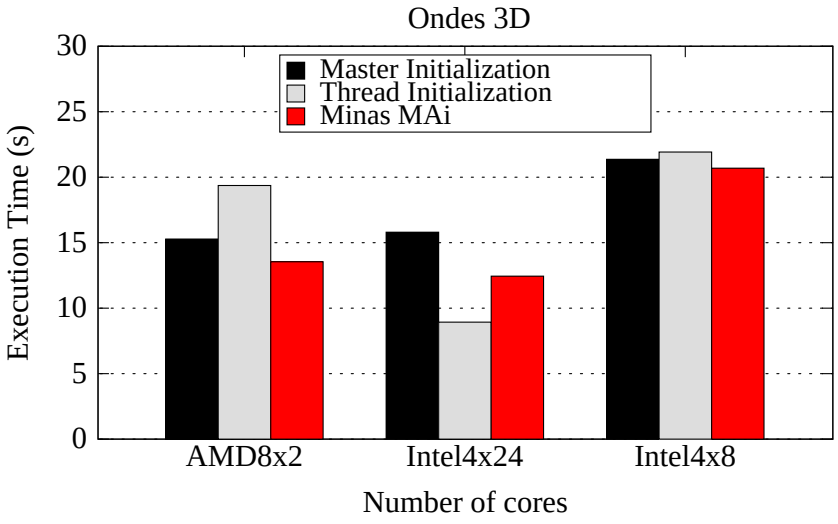


Figure 8.10: Execution Time (s) for Ondes 3D Application.

can observe that Ondes 3D application with Minas MAi has presented some slightly performance improvements compared to the other two memory affinity solutions on the AMD8x2 and Intel4x8 machines. Contrary, on the Intel4x24 machine Minas MAi has been only better than master thread initialization solution.

On the AMD8x2, the results obtained with Minas MAi and *first-touch* solutions have been very similar. Since *first-touch* and bind block (used by MAi) have similar behavior, their results are expected to be equivalent to the Minas MAi. The slight difference presented in these results comes from the usage of *cyclic* and *skew map* memory policies on some shared arrays that are accessed in read mode. It is important to mention that in the Minas MAi version we have worked only in the shared arrays. The private ones, we let Linux manage memory affinity with the *first-touch* policy.

An interesting result depicted in this figure is the one obtained with the thread initialization strategy. It has presented worse results when compared to master initialization and Minas MAi ones. Using Linux performance tools (e.g. *numactl*, *PAPI*), we have observed that Minas MAi has exploited better the memory banks and interconnection of the machine. In this case, we have used *cyclic* and *skew map* memory policies, which reduce contention problems and provide better bandwidth for data accesses. This is important for Ondes 3D on this machine (bandwidth issues), because for some steps of its computation phase, elements of the arrays are concurrently accessed by different threads.

Considering the Intel machines, since both machines have a high NUMA factor, it is important to avoid remote accesses. Therefore, in the Minas MAi version, we have used only the *bind block* function. Considering the parallelization of the application, we use rows or columns as block unit. Results obtained on the Intel4x24 shows that the best strategy is to use *first-touch* with thread initialization of data.

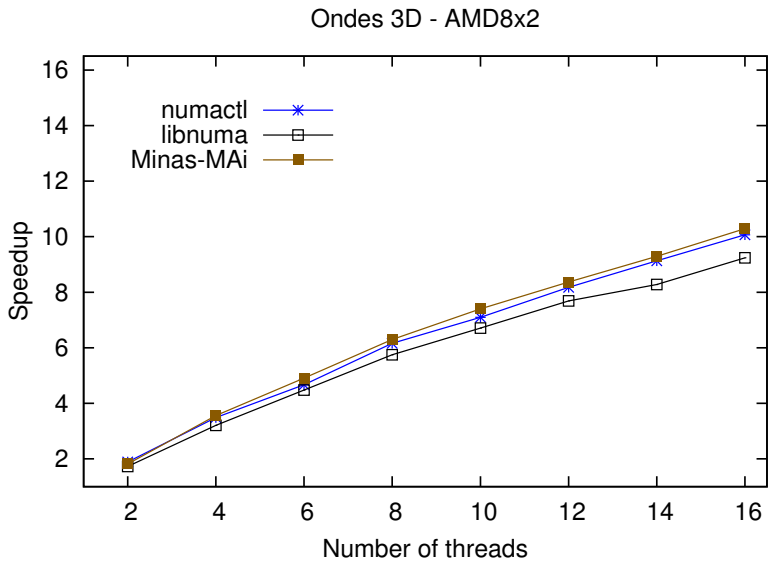


Figure 8.11: Speedups on AMD8x2 for Ondes 3D Application.

In this case, Minas MAi has split arrays in regular blocks of N rows or columns. In the initialization step, when a thread touches these blocks Minas MAi places it on the node where the thread is running. Since such blocks are aligned with memory pages, some threads may not have part of their blocks locally. We can observe that on the Intel4x8, results have been very similar. This is mainly related to its QPI (Quick Path Interconnection) network and cache memories that provides high bandwidth and low latency for remote accesses.

In Figure 8.11, we compare the speedups obtained with Minas MAi to *libnuma* and *numactl* on the AMD8x2 machine. One can notice that *libnuma* and *numactl* have had worser performances than Minas MAi. Relative average gains of Minas MAi compared to *numactl* 2% and 15% compared to *libnuma*. In the case of *libnuma*, its poor performance is related to the costs for its data allocation [Kleen 2005]. Considering the *numactl* tool, it applies the same memory policy for the whole process data. This means that *numactl* can not take memory access patterns of each array in consideration to place data whereas, Minas MAi this is possible. Results for other machines have been very similar to the AMD8x2 ones [Ribeiro 2009c, Ribeiro 2009b].

It is important to emphasize that for these results the Linux solutions demand from the user to specify which nodes and cores must be used for the application execution. In the case of *libnuma* demands considerable codification efforts, since developers must implement all data distribution algorithm and thread mapping. Additionally, for *libnuma* and *numactl* the same solution may not work on platforms with different architecture characteristics. Both *libnuma* and *numactl* do not provide architecture abstraction, which demands from the developers to describe the

machine topology in the memory affinity strategy. In the case of Minas MAi, the machine topology is automatically extracted, which provides more portability.

8.3.2 ICTM: Interval Categorizer Tesselation Model

ICTM is a multi-layered tessellation model for the categorization of geographic regions considering several different characteristics (relief, vegetation, climate, etc.). The model was first proposed in [de Aguiar 2004] and the OpenMP version of the application was proposed in [Castro 2009a].

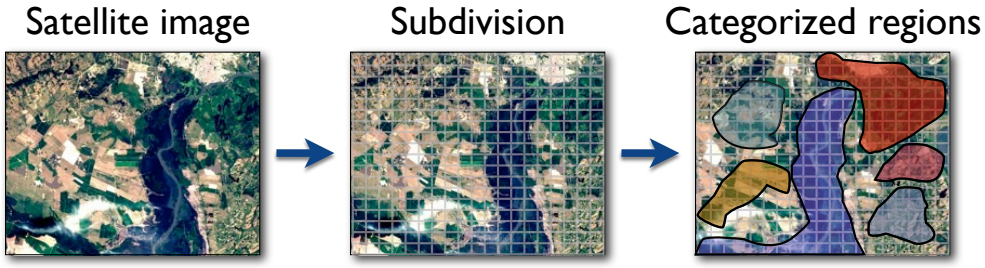


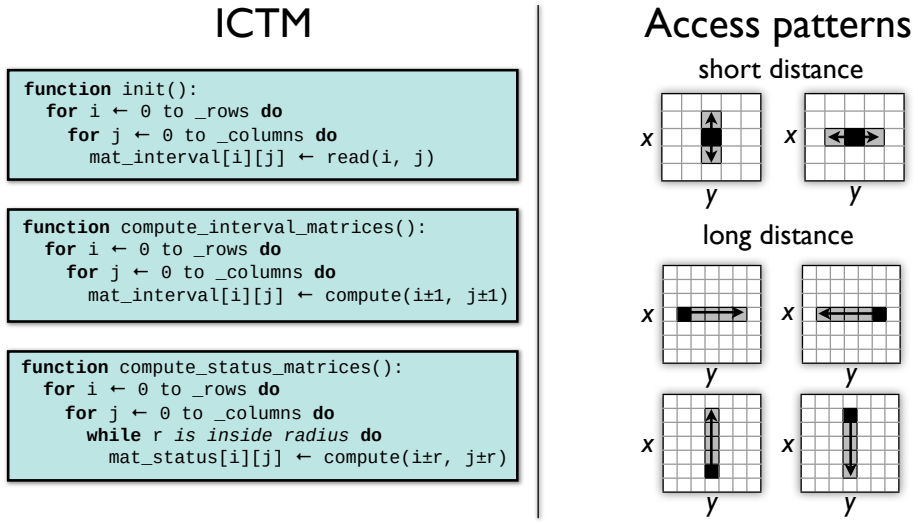
Figure 8.12: ICTM Application Input and Output.

The number of characteristics that should be studied by ICTM determines the number of layers of the model. In each layer, a different analysis of the region is performed. The input data is extracted from satellite images (Figure 8.12), in which the information is given in certain points referenced by their latitude and longitude coordinates. The extracted data is represented by a two dimensional matrix of the total area into sufficiently small rectangular subareas. In order to categorize the regions of each layer, ICTM executes five different phases. Each phase accesses specific matrices that have previously been computed and generates a new two dimensional matrix as a result of the computation. Depending on the phase, the access pattern to other matrices can either be regular or irregular [Castro 2009b].

As shown in Figure 8.13 (a), the ICTM algorithm basically uses nested loops with short and short/long distance memory accesses (Figure 8.13 (b)) during the computation phases. The short distance memory accesses are performed in immediate neighbors (one element) whereas the long distance memory accesses are performed using N neighbors, where N is defined by the model.

We carry out experiments in a problem size of 2 Gbytes and we use one thread per core of the machine. We compare Minas mechanisms (MAi and MApp) with the standard solution for memory affinity on Linux, the *first-touch*. Experiments with different problem sizes, other NUMA machines and comparisons with some other memory affinity mechanisms are presented in [Castro 2009b, Ribeiro 2009b, Ribeiro 2010b].

In order to use *first-touch* solution and Minas MAi on ICTM, we have changed application source code. In the case of *first-touch*, we included a parallel initialization of all matrices used by ICTM. Considering Minas MAi, we added some functions



(a) (b)

Figure 8.13: ICTM Application.

of MAi for data management such as *mai_ alloc()* and MAi memory policies. The results with Minas MAi were obtained by applying the most suited memory policy for each array of the application. Depending on the application phase and platform, we have chosen one of the following memory policies, *cyclic, prime mapp, skew mapp* and *bind block*. The first three memory policies are ideal for data shared in read mode over NUMA platforms that have a small NUMA factor, since they allows more throughput for data access. We have used these memory policies for the matrices used in the interval step. The latter memory policy is suitable for regular phases where threads always access the same data set. *Bind block* memory policies are also indicated for platforms with a high NUMA factor, since in this case it is important to avoid remote accesses. Since data initialization is very regular in ICTM, we have used *bind block* policy to place date in the initialization step. Figure 8.14 (a) presents the snippet of ICTM with MAi functions. In this snippet, all modifications have been done manually.

Considering the Minas MApp solution, all we have done is pre-processed the application with CUIA and then let MApp apply the modifications on the ICTM source code. MApp included in the source code some allocation functions and memory policies for all global shared arrays of ICTM. MApp did not consider the private arrays and the temporary ones created within functions of ICTM. For the global arrays and considering the platforms, MApp heuristic selected the memory policies such as *bind block* and *cyclic*. Figure 8.14 (b) presents the snippet of ICTM with MApp modifications. We can observe that differently of MAi version, this one only applies memory policies at the beginning of the application. No changes are performed between the algorithm phases.

Figure 8.15 depicts the speedups for ICTM on AMD8x2 and Intel4x8 platforms

MAi

```
mai_init();

mat_interval = mai_alloc_2D(_rows,_cols,
                  sizeof(float),FLOAT);
mat_status = mai_alloc_2D(_rows, _cols,
                  sizeof(float),FLOAT);

mai_bind_rows(mat_interval);
mai_bind_rows(mat_status);

function_init();

function_compute_interval_matrices();

//change memory policy for mat_status
mai_skew_mapp(mat_status);
function_compute_status_matrices();

mai_final();
```

(a)

MApp

```
mai_init();

mat_interval = mai_alloc_2D(_rows,_cols,
                  sizeof(float),FLOAT);
mat_status = mai_alloc_2D(_rows, _cols,
                  sizeof(float),FLOAT);

mai_cyclic(mat_interval);
mai_cyclic(mat_status);

function_init();

function_compute_interval_matrices();

function_compute_status_matrices();

mai_final();
```

(b)

Figure 8.14: ICTM Snippet with Minas: (a) MAi version (b) MApp version.

with the memory affinity solutions. One can notice that Minas mechanisms have outperformed *first-touch* solution on AMD8x2 platform. Considering the Intel4x8 platform, we can observe that Minas MAi has obtained the best results whereas Minas MApp has failed to scale for a high number of cores. However, on both platforms, the Minas mechanisms have presented better results than the *first-touch* solution.

Since ICTM has five different steps with different memory access patterns, it is difficult to take advantage of *first-touch* policy to place data over the machine. *First-touch* distributes data over the machine considering the first access on data by threads. Due to this, a different memory access pattern on any other computation step of the application can generate more remote access, contention or load balancing issues. Additionally, ICTM has one step that is dependent of bandwidth and *first-touch* only considers latency to place data over the machine memory banks. After a careful analysis of the results for each step of the application (presented in [Castro 2009a, Castro 2009b]), we observed that *first-touch* policy has obtained better results only on the phases that have regular memory accesses similar to the initialization phase.

In Figure 8.15 (a), we observe that Minas MAi and Minas MApp have generated similar performances for ICTM on the AMD8x2 machine. In this case, MApp heuristics worked well because this machine has a small NUMA factor (remote accesses are not expensive) and the *cyclic* memory policy (used by MApp) provides high bandwidth for cores. Furthermore, one can notice that in this platform, the *first-touch* fails to scale when the number of cores is increased. This platform has

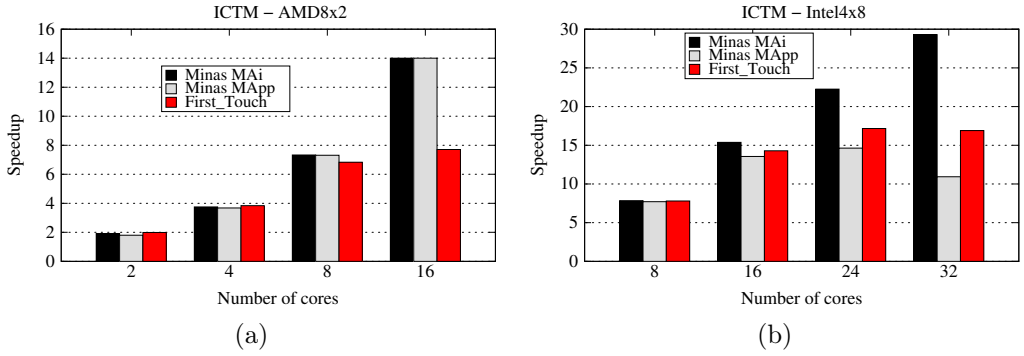


Figure 8.15: Speedup for ICTM Application on AMD8x2 and Intel4x8.

bandwidth problems and with a high number of cores being used, it is important to provide cores with throughput to move data. Figure 8.15 (b) shows that Minas MAi mechanism has obtained good performance and scalability for all number of cores. Minas MApp have not performed well on the Intel4x8 because its heuristic only consider standard *cyclic* and *bind block* memory policies. It does not make use of other cyclic memory policies such as *skew mapp/cyclic neighbors* (used in Minas MAi solution).

8.4 Conclusions

In this chapter, we presented the performance evaluation of Minas framework on OpenMP benchmarks and applications over different NUMA machines with multi-core chips. We used a synthetic benchmark, some representative benchmarks (e.g. Stream Benchmark and NAS Paralllel Benchmarks) and two real applications from geophysics domain. Furthermore, we compared Minas results to standard solutions for memory affinity on the Linux operating system.

Our experiments show that Minas improves performance of OpenMP benchmarks and applications, compared to the Linux and GCC/ICC compiler solutions. Gains with both Minas approaches were observed for applications over the three NUMA platforms used in our experiments. Results presented in this chapter show that memory affinity must be efficiently managed in NUMA machines and that this management should be simplified for the developer. Therefore, a framework that helps developers to do this is essential.

Another important result is the code and performance portability provided by Minas framework. Considering code portability, Minas is capable of extract the machine topology and use this information at runtime to place data and threads on the machine. Considering the Linux solutions, the user have to specify which nodes and cores should be used for the application execution. Additionally, in Linux only the *libnuma* solutions provides support to different memory policies within the same parallel application. The performance portability of Minas is due to its capacity to

adapt to different NUMA machines and provide improvement gains for OpenMP applications.

In the next chapter, we show the performance evaluation of Minas framework components in two other parallel environments, the Charm++, AMPI and OpenSkel. The performance evaluation is performed using some benchmarks developed with Charm++, AMPI and OpenSkel.

Chapter 9

Evaluation on High Level Parallel Systems Benchmarks

In this chapter, we present the performance evaluation of Minas framework components inside the three parallel systems, Charm++, AMPI and OpenSkel. For the performance evaluation, we use benchmarks developed with the Charm++, AMPI and OpenSkel parallel systems and the three NUMA platforms described in chapter 7. We consider benchmarks with different memory access characteristics to better explore the memory affinity management of the Minas framework. Similar to the performance evaluation on OpenMP benchmarks, our objective is to analyze the impact of different architectures in the memory affinity management for parallel applications. Minas results are compared to the ones obtained with the original version of each parallel system. We first describe each one of the benchmarks used in our performance evaluation and then, we present their results and analysis.

9.1 Charm++ Benchmarks

In this section, we present the performance evaluation of the two memory affinity mechanisms developed for Charm++ parallel system using Minas components. In our performance evaluation we use three benchmarks from Charm++ examples, Kneighbor, Molecular 2D and the Jacobi 2D [PPL-Charm++ 2011]. The Kneighbor and Molecular 2D benchmarks allow us to evaluate Minas MAi data placement (+**maffinity**) for applications with different characteristics (memory consumption and communication) and requirements (memory bandwidth and latency). The jacobi 2D benchmark is used to evaluate the performance of our NUMA-aware load balancer, since this benchmark presents load unbalance characteristics. Additionally, jacobi 2D has important ratio of communication and computation. For this evaluation, we use the three NUMA platforms described in chapter 7 and the Charm++ 6.2.0 with the multi-core installation.

9.1.1 Memory Policies

For the results presented in this section, we have used one thread per core on all executions. We compare the results obtained with the +**maffinity** mechanism to ones obtained by the use of +**setcpuaffinity** mechanism [Mei 2010].
Kneighbor: it is a synthetic iterative benchmark that performs mostly communication operations, no important computation is performed. In this benchmark, a

matrix of one dimension is used to store chares. In each iteration of Kneighbor, a chare sends messages to its k neighbors. A iteration finish, when the destination chares receive their messages. In our experiments, we have used the number of cores as number of chares, k is equal to 3, 100 iterations and message sizes from 16 bytes to 4096 bytes.

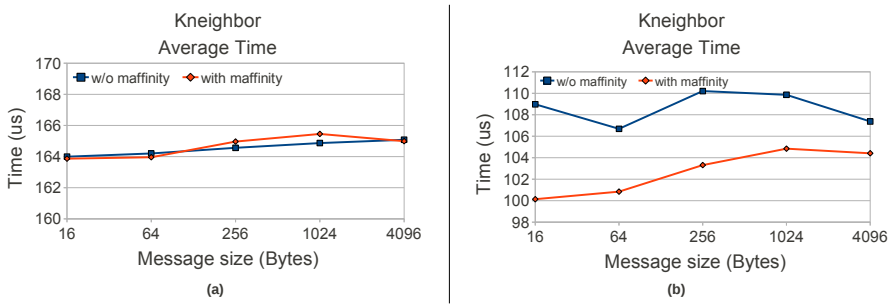


Figure 9.1: Execution Time (us) of Kneighbor: (a) AMD8x2 (b) Intel4x8.

Figure 9.1 shows the average time for Kneighbor benchmark on the AMD8x2 and Intel4x8 platforms for different sizes of messages. We can observe that for Kneighbor benchmark the memory affinity support has obtained better results on average for both machines with different message sizes. However, gains are more relevant in the Intel4x8 NUMA machine (up to 8%). In this machine the NUMA factor is higher and consequently, the impact of improving memory affinity is more significant. For this benchmark and considering the selected NUMA machines, it is important to reduce the latency for cores to get data. To do so, we have fixed threads using the cpu affinity support of Charm++ and then, we have applied the **+maffinity** to place data on the node where threads are running. Therefore, performance has been improved due to the better data locality.

Figure 9.2 reports the average time for Kneighbor benchmark when executed on the Intel4x24 platform. For these experiments we used 24 cores (one NUMA node) and 64 cores (four NUMA nodes) and messages sizes of 1024 bytes. We can observe that results with memory affinity support are similar to without affinity for 24 cores. In this case, the application runs in only one NUMA node (set threads with **+setcpuaffinity**) and consequently, any improvement can be generated by the use of **+maffinity** since data is already local to threads. However, for 64 cores we notice an improvement gain up to 6%. Since more NUMA nodes are used to execute the application, the **+maffinity** support is able to spread application data over the memory banks of the machine in order to reduce the NUMA penalties.

Molecular 2D: it is a benchmark from Charm++ examples that computes biomolecular simulations using molecular dynamics. In each step of the benchmark, the properties of each particle (acceleration, velocity and position) and the iteration between particles are computed. In our experiments, we have used 50 steps and a 25 patches on the AMD8x2 and Intel4x24 and 16000 patches on the Intel4x8 to analyze the impact of data sizes that do not fit on cache memories.

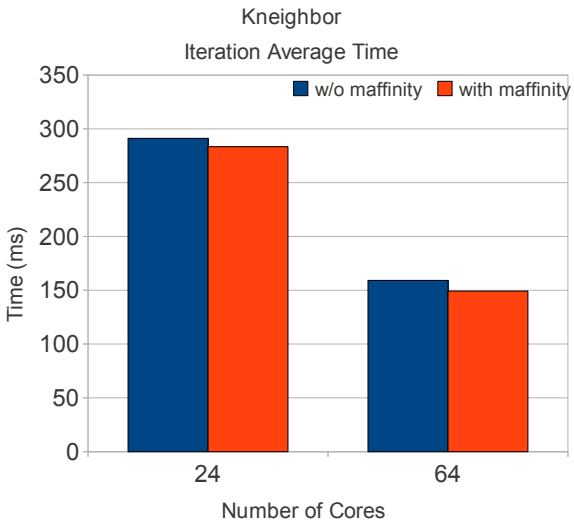


Figure 9.2: Execution Time of Kneighbor on Intel4x24.

In Table B.2, we present the execution time for a step (ms) of Molecular 2D benchmark when executed with +**setcupaffinity** and with +**maffinity** options. On general, +**maffinity** has presented some slight improvements when compared to +**setcupaffinity** option (up 5% of gains). The best result has been obtained on the Intel4x8 NUMA machine when four nodes (32 cores) of the machine have been used. The iteration time difference is mainly related to the reduction of concurrent remote accesses performed by threads on the same memory bank. The use of *cyclic neighbors* memory affinity strategy has provided better load balancing considering memory pages distribution on physical memory banks. Consequently, more physical memory pages were available for the concurrent access performed by threads.

Table 9.1: Execution Time (ms) of One Step of Molecular 2D

	AMD8x2		**Intel4x8**	
	8 Cores	16 Cores	16 Cores	32 Cores
w/o maffinity	131.46	68.84	1083.74	698.67
maffinity	125.08	67.43	1038.80	692.06

Figure 9.3 shows the time per step for the Molecular 2D benchmark on the Intel4x24 platform. In these experiments we used 24 cores (one NUMA node) and 64 cores (four NUMA nodes). The +**maffinity** results for both number of cores have been better than the ones obtained without the memory affinity support (up to 12% of gains). Considering the benchmark characteristics, we have applied the *cyclic neighbors* memory policy to distribute its data over the machine memory

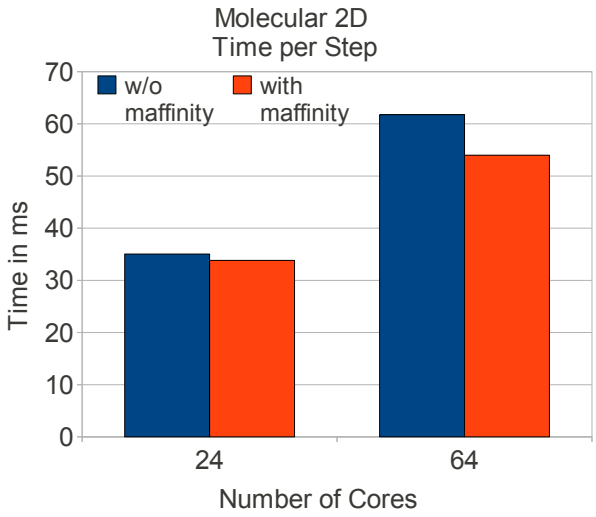


Figure 9.3: Iteration Average Time of Molecular 2D on Intel4x24.

banks. This memory policy adapts to the Molecular 2D default work distribution, which is a round-robin strategy. More results with the **+maffinity** support are presented in the work [Ribeiro 2010d].

9.1.2 NUMA-aware Load Balancer

In this section, we present the performance evaluation of the proposed NUMA-aware load balancer. We compare the results obtained with **NumaLB** to the ones obtained with no load balancer and with two other load balancers, the Metis [Karypis 1995] and the Greedy [PPL-Charm++ 2011].

Similar to NumaLB, the Metis load balancer also exploit the communication costs to balance the load between the machine cores. It uses a graph partition mechanism from the Metis library to create a schema that represents the threads communication. Contrary to this strategy, the Greedy load balancer does not consider any communication characteristics of the application to perform load balancing. Its strategy places the heaviest chare on the less loaded processor, until the load balancing is reached. Using Jacobi 2D benchmark, we first depict the overall performance obtained with NumaLB and compare it to the results obtained with other load balancers. After that, we present some statistics of the NumaLB that allow us to evaluate the overhead to extract the machine topology with numArch.
Jacobi 2D: it is an unbalanced benchmark from Charm++ examples that performs a 2D stencil computation. This benchmark iterates over a data set, updating elements until some condition is reached. The elements are updated with the average of its current value and the current values of its four neighbors. In our experiments, we have used 36 chares of 6x6, a chunk size of 64 elements and 10 iterations [Jacobi 2011].

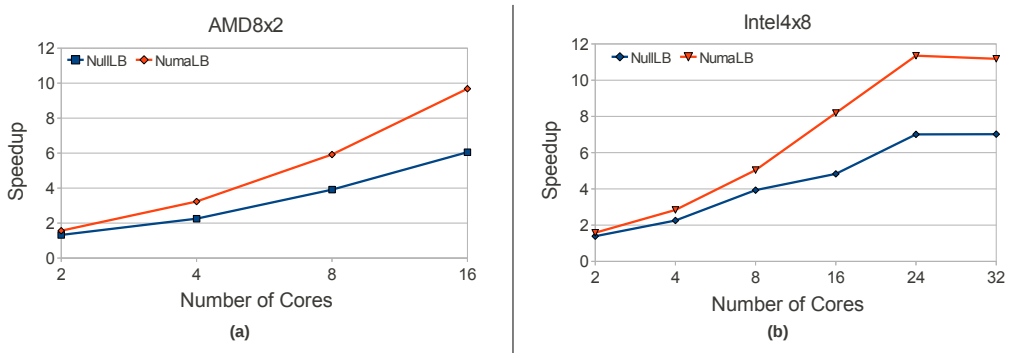


Figure 9.4: Jacobi 2D Speedups with NumaLB Load Balancer: (a) AMD8x2 (c) Intel4x8.

Figure 9.4 shows the speedups for the Jacobi 2D with and without the NumaLB load balancer. Our experimental results with Jacobi 2D have shown that the NumaLB load balancer achieve performance improvements of up to 68%, with an average of 24%, over the no load balancer version of Jacobi 2D on two NUMA multi-core machines. We can observe that on both machines, Jacobi 2D benchmark has presented better scalability with the NumaLB load balancer. In the NumaLB case, objects migrations considering the NUMA factor avoids long distance communication between chares. Due to this, the impact of the NUMA design on the application execution is reduced.

We have observed the same behavior on both machines, however improvement gains are more significant in the Intel4x8 because it presents a higher NUMA factor. Since speedups have been computed considering the Jacobi 2D execution time, they allow us to confirm that the overhead to extract the machine topology with numArch does not degrade the overall performance of the benchmark. Additionally, these results also allow us to confirm that the NumaLB strategy does not take too many time to be computed. Therefore, the load balancing is performed without an important impact on the overall application execution time.

Figure 9.5 reports the speedup obtained with NumaLB, MetisLB and GreedyLB for Jacobi 2D benchmark. Overall, NumaLB strategy has performed up to 16% better than the MetisLB balancer. Despite MetisLB considers the communication characteristics of the application to perform the load balancing, it does not take into account the machine topology on its strategy. MetisLB can not exploit the access latency costs of a NUMA platform, performing worse than NumaLB on both machines. Opposed to the comparison with MetisLB, when compared to the GreedyLB load balancer, NumaLB has presented similar performances. In the case of the AMD8x2 machine, this is due to the small NUMA factor of the machine. Considering the Intel4x8 machine, it has a huge shared L3 cache memories that favors the GreedyLB load balancer. The shared L3 caches on the Intel4x8 machine reduces the costly memory access to local and remote DRAMs. Even though, NumaLB load balancer

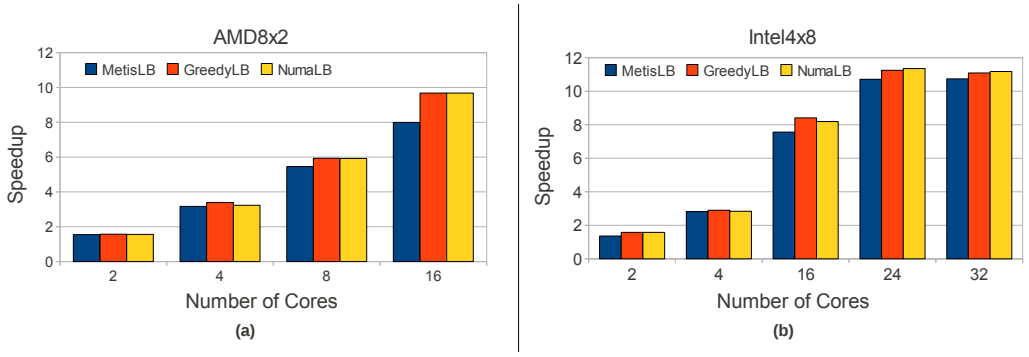


Figure 9.5: Jacobi 2D Speedups with Different Load Balancer: (a) AMD8x2 (c) Intel4x8.

has not decreased the performance of Jacobi 2D and for large number of cores it has presented better results.

Table 9.2: Execution Statistics of Load Balancers.

	AMD8x2		**Intel4x8**	
	Init Time (s)	Strategy Time (ms)	Init Time (s)	Strategy Time (ms)
MetisLB	0.22	0.502	0.106	0.549
GreedyLB	0.17	0.055	0.101	0.063
NumaLB	0.19	0.301	0.100	0.264

The main contribution of this thesis in the design of NumaLB load balancer is the representation of the NUMA machine hierarchy and topology provided by Minas numArch module. Therefore, it is also important to evaluate the overhead imposed by numArch to extract the machine information. To do so, we selected two performance metrics the initialization time to load the load balancer information and the time taken by the load balancer to perform its strategy. Table B.3 reports each of these metrics for MetisLB, GreedyLB and NumaLB. We can observe that the time spent by NumaLB to initialize the load balancer is 13% shorter than the MetisLB one and 10% longer than the GreedyLB one. Considering the time taken to computed the NumaLB strategy, it has been better than MetisLB and worse than GreedyLB. As mention before GreedyLB does not consider any communication information from the application to perform load balancing. It only consider the load of each processor. Therefore, its strategy is less expensive to be computed than MetisLB and NumaLB ones. However, this difference does not reduce the overall performance of NumaLB.

9.2 AMPI Benchmark

In order to evaluate the performance of the NUMA-aware Isomalloc on AMPI, we have selected as benchmark the Jacobi3D (performs 3D stencil computation) from AMPI examples. In each iteration of Jacobi 3D, there is a *AMPI_ migrate* call that signals the adaptive runtime system to perform virtual processes migration. Depending on the strategy some virtual processes migration will happen. For our experiments, we consider the Charm++/AMPI 6.2.0 version with net-linux-x86_64 installation. The analysis of the results compares the performances obtained with NUMA-aware isomalloc policies to the ones obtained with the original isomalloc.

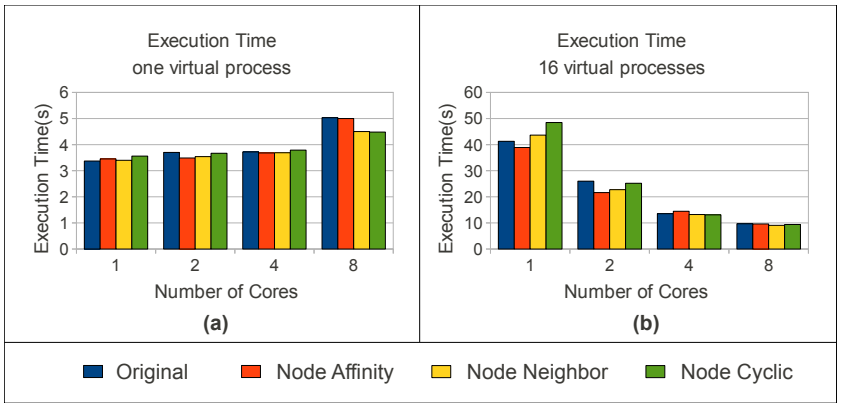


Figure 9.6: Jacobi 3D Benchmark Execution Time on AMD8x2.

In Figure 9.6, we present the execution time for Jacobi benchmark with Isomalloc (original) and the NUMA-aware Isomalloc (*node affinity, node neighbor* and *node cyclic*) for the AMD8x2 NUMA machine.

We can observe that when using only one virtual process the performance with the different versions of isomalloc have been similar (Figure 9.6 (a)). On general, NUMA memory policies have been used to correct any data placement mistake (*node affinity* policy) or to provide more bandwidth for the application (*node cyclic* and *node neighbor*). In some experiments, we can observe that the NUMA-aware isomalloc implementation has presented some improvement gains (up to 6% execution time with 8 processes). This is due to the mapping that these memory policies have performed when *AMPI_ migrate()* function has been called in the application. Considering this machine, *node affinity* memory policy has presented better results for a small number of processes, because it allows more data locality for the processes. Contrary to *node affinity*, the memory policy *node cyclic* has presented better results for a large number of processes. Since *node cyclic* spread data over the machine memory banks, it avoids memory contention and provides better memory bandwidth when a higher number of processes and virtual processes is used.

Figure 9.6 (b) presents the execution time for executions of Jacobi benchmark with sixteen virtual processes. We can observe that the NUMA-aware isomalloc has

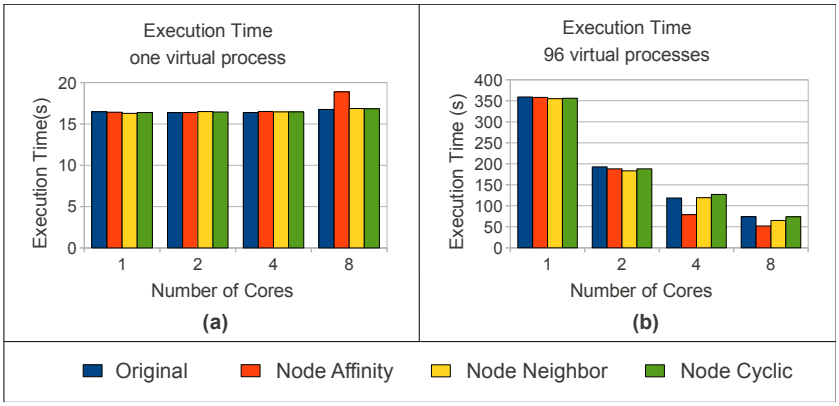


Figure 9.7: Jacobi 3D Benchmark Execution Time on Intel4x24.

obtained, on average, the best results (up to 16% with *node affinity* policy). Due to the high number of virtual processes, when the migration function is called on the benchmark, more corrections on data placement are performed by NUMA-aware isomalloc. Similar to the experiments with one virtual process, *node affinity* memory policy has presented better results for small number of process and *node cyclic* has been more performant for a large number of processes.

Figure 9.7 reports the the execution time for Jacobi benchmark with Isomalloc (original) and the NUMA-aware Isomalloc for the Intel4x24 NUMA platform. Figure 9.7 (a) and (b) presents the execution time that we have obtained with the different implementations of isomalloc.

The results obtained on the Intel4x24 machine for one virtual process do not present any improvements gains when compared to the original isomalloc. The Intel4x24 machine has less NUMA nodes than the AMD8x2 one. Consequently, the NUMA impact in the migration of virtual process is smaller when only one virtual process per core is used. However, by increasing the number of virtual processes (Figure 9.7 (b)), we can observe that NUMA-aware isomalloc results present significant improvement gains for execution time (up to 33%). In this case, more virtual process migrations are performed by the runtime system in order to keep load balancing among the machine cores. In such scenario, the memory policies are used to place the virtual processes data on the new NUMA node. Particularly, the memory policy *node affinity* has presented better results, since it ensures that data is placed in the same NUMA node where the virtual process is running.

9.3 OpenSkel Version of Stamp Benchmark

In order to evaluate the performance of the NUMA support provided by the integration of Minas MAi memory policies on OpenSkel System, we have selected three benchmarks from STAMP [Minh 2008] (intruder, kmeans and vacation). We use the OpenSkel version of these benchmarks, which are presented in [Góes 2010a,

Góes 2010b]. All selected applications have been executed with the recommended input data sets. Kmeans and Vacation have two input data sets, high and low contention. As Intruder only has high contention input data sets, we chose the low contention inputs for Kmeans and Vacation to cover a wider range of behaviors. Table 9.3 summarizes the main characteristics of these applications. A detailed description of the application computation is presented later on in this section.

Table 9.3: Summary of STAMP application runtime characteristics.

Application	Scalable up to # Cores	Contention on Worklist
Intruder	8	high
Kmeans	4	medium
Vacation	8	high

For this performance evaluation, we used TinySTM as Software Transactional Memory (STM) with eager as policy for conflict detection [Felber 2008]. The conflict detection policy is responsible to deal with concurrent accesses on the same data within the STM system. In STM, the eager policy tries to solve conflicts when they happen whereas the lazy policy only solves them at the end of the transaction. We selected the eager policy because it suffer more from the NUMA penalties, as presented in [Góes 2011]. We now describe the selected benchmarks used in our performance evaluation and then, results are presented for each machine/benchmark. The analysis compares the performance obtained with OpenSkel+Minas to the ones obtained with the original OpenSkel implementation. In the original version of OpenSkel the memory affinity, thread and data placement, is performed by the operating system, whereas in the Minas version both data and thread placement are performed by the OpenSkel runtime system using the numArch and MAi components of Minas.

Intruder: it is an application that implements a signature-based network intrusion detection system (NIDS). It has three phases and it uses dynamic data structures to represent data (tree). In th first phase, the application reads packets from a queue. After that, these packets are computed by threads using a shared tree to represent them. In the last phase, string search is performed to match signatures.

Figure 9.8 reports the speedups that have been obtained with Intruder on the AMD8x2 and Intel4x8 machines. Results are presented for the original version of OpenSkel runtime and with Minas memory support on the OpenSkel runtime system. On the AMD8x2 machine, OpenSkel+Minas has obtained the best performances due to its better data placement over the machine. OpenSkel+Minas considers the characteristics of both machine and application in order to avoid high latencies and to allow good bandwidth to access data. However, on the Intel4x8 machine, one can observe that OpenSkel+Minas has presented some slight improvements on speedups compared to the original version of OpenSkel. In the Intel4x8 machine remote data requests rarely leads to an access to a remote memory bank. It comes from the fact that each node has a large shared L3 cache and they are

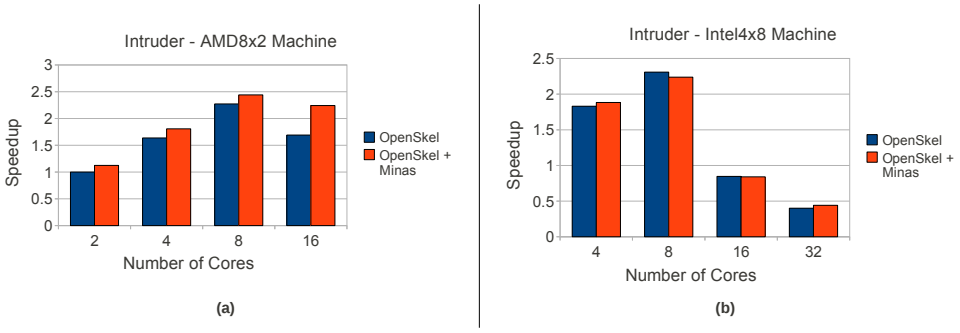


Figure 9.8: Speedups for Intruder Application: (a) AMD8x2 (b) Intel4x8.

interconnected through high speed communication channels. Instead of accessing a remote memory bank directly on a data request, a core checks if the data is available on its local and remote caches. Due to this, the impact of memory policies in this machine is smaller.

Kmeans: it is an iterative application that performs a clustering algorithm to group elements with some similarity into K clusters. The clusterization is based on the distance between elements and their centroids. The master thread is responsible for compute the distances and centroids at the end of each iteration. In the original version, arrays are used to save elements and threads compute chunks of the array. In the OpenSkel version a Worklist is used to store elements.

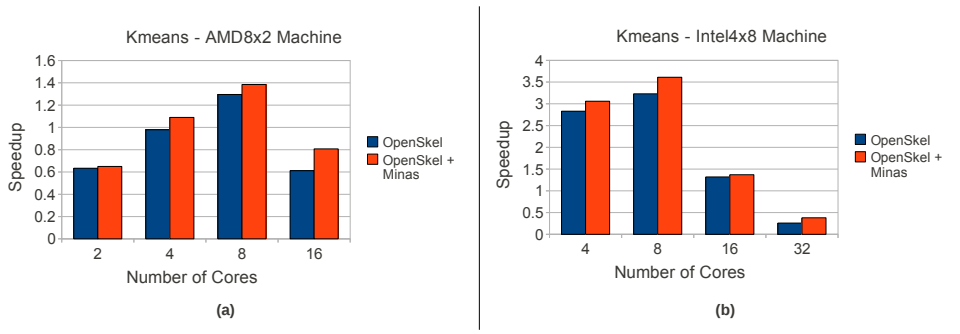


Figure 9.9: Speedups for Kmeans Application: (a) AMD8x2 (b) Intel4x8.

In Figure 9.9, we depict the speedups that have been obtained with Kmeans application on the AMD8x2 and Intel4x8 machines. Results are presented for the original version of OpenSkel runtime and with the Minas memory support on its runtime system. On general OpenSkel+Minas has obtained the best performances on both platforms. This is due to Minas data placement over the machine memory banks. In the case of Kmeans that has medium contention as main characteristic, we have chosen to bind data closer to threads when a small number of threads is used. Considering the speedups for a larger number of threads, the selected

memory policy to place data was *cyclic neighbor*. In this case even with medium contention, it is necessary to provide good memory bandwidth for cores. One way to this is by spreading data over different memory banks and guarantee that more interconnection network links will be available to get some data.

Vacation: It is an application that emulates an on-line travel reservation system. In the application each client has a number of requests that has been generated random way. Each transaction computes one request and makes some accesses to a database server. The main difference from the original implementation to the OpenSkel one is that the second, authors have created a worklist and transformed the requests into work-units.

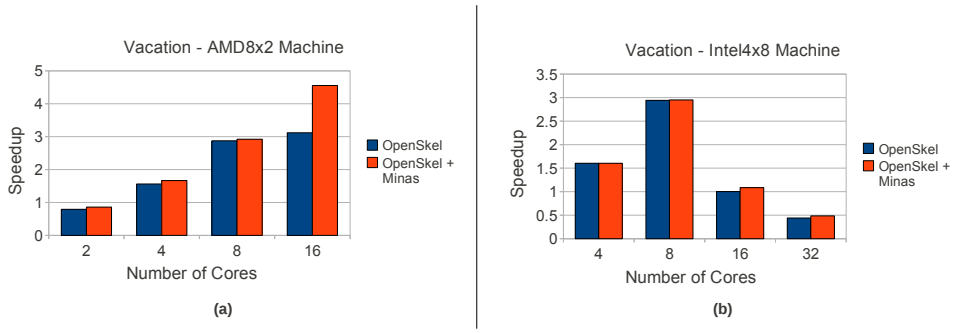


Figure 9.10: Speedups for Vacation Application: (a) AMD8x2 (b) Intel4x8.

Figure 9.10 shows the speedups that have been obtained with Vacation on the AMD8x2 and Intel4x8 machines. Results are presented for the original version of OpenSkel and with Minas support for memory affinity. On general, for the AMD8x2 machine, Minas has provided better performances for OpenSkel whereas on the Intel4x8 machine results with and without Minas support have been very similar. In the case of AMD8x2 machine when using Minas support, we can observe that the application still scaling when the number of cores is increased. This scenario is not true on the Intel4x8 machine, where the application fails to scale with more threads. However, Minas data placement allows some slight performance improvements for such scenario. Considering the application characteristics, the high contention on some data sets generates false sharing in cache lines. Since Minas does not deal with false sharing, its improvements are limited to the global memory affinity.

9.4 Conclusions

In this chapter, we presented the performance evaluation of Minas memory affinity mechanisms on different benchmarks developed with high level parallel languages and interfaces. Additionally, we have performed a number of experiments with various benchmarks and NUMA platforms. These experiments have allowed us to show the performance improvements that Minas can bring to the three parallel systems

by using its components to place data, map threads and extract the machine architecture.

Considering the Charm++ system and the selected platforms, we have observed improvement gains of up to 12% when compared to the original version. In the case of AMPI, where communication is important, some significant improvements have been observed when Minas memory policies were used (up to 33%). Slight performance improvements have been obtained for OpenSkel in the Intel4x8 machine whereas, in the AMD8x2 platform performance improvements have been significant. This is mainly related to cache memory sizes that in the case of AMD8x2 machine is very small, which makes main memory utilization and memory affinity management much more important.

We can then conclude that on such parallel systems, the memory access costs of NUMA platforms have a significant impact on their application performances. The best data distribution depends on the parallel system runtime implementation, the application characteristics and the machine architectures. We have observed that on NUMA machines with high NUMA factor, the best memory affinity strategy is generally to place threads and data closer. This can be reached by placing threads and their data in the same NUMA node and by performing data migration during the application execution. In contrast, the NUMA machines with low NUMA factor require a better usage of the machine memory banks in order to reduce load balance and memory contention issues. Additionally, application characteristics such as data sharing level among threads (communication patterns) are also determinants factors in the selection of the best memory affinity strategy. Therefore, it becomes crucial to have a portable and flexible solution such as Minas framework that can match the architecture and application characteristics to manage memory affinity on NUMA machines.

Minas is easily integrated in parallel environments, since it offers high level interfaces to deal with memory affinity. This framework provides to parallel environments a NUMA machine representation which allows them to abstract the machine architecture and topology. NumArch module provides a high level library that can be used in runtime systems to get the machine information. Such information allows parallel environments to be portable over different multi-core NUMA machines, since the architecture abstraction is ensured. Additionally, Minas framework has an interface (MAi) that supports several memory policies to perform data placement with different granularities. These memory policies are designed in a such way that they can easily integrated in different contexts such as Charm++, AMPI and OpenSkel to enhance memory affinity.

Chapter 10

Conclusions and Perspectives

In this last chapter, we conclude the thesis presenting its main objectives, the contributions on the memory affinity management for NUMA machines and the perspectives generated by this work.

10.1 Thesis Objectives

In the past few years, multi-core chips have become a trend in the processor design of shared memory machines. However, as the number of cores per chip increases, memory access to the shared memory becomes a major bottleneck. Multiple cores access the same shared memory resulting in memory contention and a worse than expected performance from parallel machines. To alleviate the memory problem, Non-Uniform Memory Access (NUMA) designs have been employed in multi-core machines.

Modern NUMA multi-core machines are generally built with on-chip memory controllers and an associated memory bank for each socket. The controllers are interconnected by a special network, thus allowing access to physically remote memory banks. This design reduces memory contention, if compared to a conventional shared memory machine, while still providing the abstraction of a single shared memory all the cores. However, this design can potentially increase the memory access latency and degrade bandwidth usage if the interconnection network itself becomes congested. Thus, the adoption of an efficient memory affinity strategy becomes crucial to achieve good performance in such machines. In NUMA machines, memory affinity mechanisms try to keep data close to the core that accesses them in order to reduce the remote memory access costs for parallel applications.

Considering shared memory machines, there are several programming languages that can be used to develop parallel applications. For instance, the OpenMP API and Charm++ parallel language are examples of programming interfaces. However, most parallel language compilers do not address multi-core machines with NUMA design, lacking support to control memory affinity. That task ends up being handled by the operating system or even worse by the programmer. The work developed in this thesis is related to the memory affinity management for parallel languages on multi-core machines with NUMA design. The main objective is to hide from the programmer the complexity of controlling the memory affinity within a parallel application source code. The NUMA platform characteristics and the application memory access patterns are used in order to provide a fine grain of memory affinity management.

10.2 Contributions

To enhance memory affinity on multi-core machines with NUMA design, we proposed the Minas framework (chapters 4 and 5). This framework deals with memory affinity by controlling data and thread placement on NUMA platforms. The data and thread placement mechanisms rely on the machine and the application characteristics. The machine topology and hierarchy representation provied by Minas enables it to address the hierarchical architecture of these machines. The Minas preprocessor acts in the extraction of the application characteristics that must be considered to place data. This information is coupled with a number of different memory policies and thread mappings inside Minas to improve memory affinity.

Minas implements two types of memory policies: the *bind* ones, that reduce the access latency perceived by threads to get data, and the *cyclic* ones that reduce memory contention, providing more bandwidth for threads to get data. These memory policies can be applied in different levels of the application data such as the heap and variables. Considering the thread mapping mechanisms inside Minas, the framework implements static and dynamic mechanisms. The static one places threads over the machine cores with the objective of maximizing cache sharing between them. This is achieved by Minas using the machine topology and memory traces of the application. The dynamic thread mapping is implemented as NUMA-aware load balancer that uses the machine NUMA topology and the application communication characteristics to map threads at runtime.

In Minas framework, we showed that the memory affinity management can be done explicitly, by the programmer, or in an automatic way with use of the MApp preprocessor. Developers who know their application can manually control the memory affinity providing some hints about memory access patterns for Minas. In this case, the programmer explicitly modifies the application source code, using MAi interface to allocate and place data. Using these functions, at runtime Minas maps application data over the NUMA machine nodes. Developers who do not have a priori knowledge about memory access patterns of the application can automatically control affinity by using Minas MApp. This mechanism transforms the application source code using characteristics from both the machine and the application, extracted at compilation time. We also showed that both mechanisms can be combined in order to enhance even more the memory affinity and consequently, the performance of a parallel application.

We employed Minas framework components in four parallel interfaces/languages used to develop parallel applications, applying the components at different levels. First, at the language level, we have showed that Minas can be used at compile time to control memory affinity for OpenMP applications. This support is compiler independent, which enhances the applicability of Minas framework. Then we explored Minas framework to control memory affinity for dynamic applications such as the ones developed with the parallel system Charm++ and the AMPI. In the case of Charm++ and AMPI, an integration of Minas components was implemented for their runtime system, which provides users with a transparent memory affinity sup-

port. Finally, in the algorithm level, we integrated Minas to a skeleton framework, OpenSkel, to provide it with memory affinity support. Using the information provided by the skeleton, Minas is capable of dealing with data and thread placement for a skeleton based application.

10.3 Perspectives

The memory affinity approaches proposed in this thesis leads to a number of perspectives.

Enhance the NUMA Machine Model: The current model of the NUMA machines used to represent the core topology and the memory hierarchy can be enhanced to include even more details about the architecture. For instance, the memory hierarchy representation relies on the main memory read latency and bandwidth measurements. An improvement would be the use of write operations as well, to obtain those metrics. In this case, our memory hierarchy representation would be capable of modeling the different type memory accesses that an application can perform. Considering the bandwidth, our model could use multiple threads in the benchmarking step to compute the memory bandwidth when memory contention is present. This information would allow Minas framework to improve its heuristic to map data taking into account the saturation of interconnection network. Another possible evolution is the use of latency and bandwidth for cache memories to provide Minas with overhead associated with intra-node communication. The use of hardware counters to retrieve the application information at runtime is also a possible improvement for our model. The machine core topology could be enhanced to support operating systems other then Linux. Another possibility is to have our model integrated in user level tools such as the *hwloc* library.

Get More Information About the Application: Minas framework automatic tuning of parallel applications could be improved to support more languages and interfaces. In this thesis, our results with the automatic approach only considered applications written in C with OpenMP API. In the case of OpenMP, one perspective is to extend our preprocessor to also support Fortran based applications. Many OpenMP parallel benchmarks and applications are written in Fortran. Therefore, a support for this language is important in the context of high performance computing. In this case, it is necessary to extend our preprocessor with the Fortran grammar in order to retrieve the application information. Considering the OpenMP applications, information such as memory access on variables inside the parallel regions should be considered. This information would allow Minas framework to better deal with data distribution over the machine. For instance, indirect accesses or sequential ones provides Minas with the information on how data is accessed inside a parallel region at compile time. In this context, compile time automatic support for other parallel programming systems such as Charm++/AMPI are also a perspective of this thesis. Additionally, Minas can also profit of runtime information from the application to correct any data and thread placement.

Support for Hybrids Architectures: Multi-core architectures are becoming more and more complex. Besides the NUMA design, current multi-core machines also feature complex cache memory hierarchies. Additionally, they can also be equipped with GPUs (graphics processing units) to increase their processing power. Although the GPU brings more performance for the multi-core machines, it comes with the cost of a more complex architecture. Thus a memory affinity support for these machines becomes necessary to reduce the communication costs between GPUs and cores. Therefore, one possible work is to extend Minas framework to support hybrid platforms. Minas mechanisms to deal with memory allocation, data placement and thread scheduling should be aware of the different elements to enhance memory affinity for the parallel applications. The machine model representation need to be extended to consider such a heterogeneous machine. Our mechanisms to deal with memory management should consider the different memory spaces (i.e. GPU memory and machine memory) that composes the global memory and the different representations of data structures in GPU memory and machine memory. Additionally, in high performance computing the number of cores per processors will increase, leading to the many-core architectures. Therefore, multiple levels of non-uniform access in the memory hierarchy are expected. Providing support for these architectures is another perspective work created by this thesis.

Bibliography

[Agarwal 1995] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie and Donald Yeung. *The MIT Alewife machine: architecture and performance*. SIGARCH Comput. Archit. News, vol. 23, no. 2, pages 2–13, 1995. 11, 12

[Agarwal 2006] Tarun Agarwal, Amit Sharma and Laxmikant A. Kalé. *Topology-aware task mapping for reducing communication contention on large parallel machines*. In 20th International Parallel and Distributed Processing Symposium – IPDPS, 2006. 39

[AMD 2011a] AMD. *Advanced Micro Devices - AMD Opteron*. `http://www.amd.com`, 2011. 17, 18

[AMD 2011b] AMD. *AMD64 Architecture Programmer's Manual Vol 2 System Programming*. `http://support.amd.com/us/Processor_TechDocs/24593.pdf`, 2011. v, 18, 19

[Antoniu 1999] Gabriel Antoniu, Luc Bougé and Raymond Namyst. *An Efficient and Transparent Thread Migration Scheme in the PM2 Runtime System*. In Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, pages 496–510. Springer-Verlag, 1999. 97

[Awasthi 2010] Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian and Al Davis. *Handling the problems and opportunities posed by multiple on-chip memory controllers*. In Proceedings of the 19th international conference on Parallel architectures and compilation techniques, PACT '10, pages 319–330, New York, NY, USA, 2010. ACM. 15, 41, 179

[Benkner 2002] S Benkner and T Brandes. *Efficient Parallel Programming on Scale Shared Memory Systems with High Performance Fortran*. Concurrency: Practice and Experience, vol. 14, pages 789–803, 2002. 40, 180

[Berger 2000] Emery D. Berger, Kathryn S. McKinley, Robert D. and Paul R. Blumofe Wilson. *Hoard: a scalable memory allocator for multithreaded applications*. In Proceedings of the ninth international conference on Architectural support for programming languages and operating systems, ASPLOS-IX, pages 117–128, New York, NY, USA, 2000. ACM. 30, 42, 59

[Bhatelé 2009] Abhinav Bhatelé, Laxmikant V. Kalé and Sameer Kumar. *Dynamic topology aware load balancing algorithms for molecular dynamics applications*. In Proceedings of the 23rd international conference on Supercomputing, ICS '09, pages 110–116, New York, NY, USA, 2009. ACM. 39

[Bircsak 2000] John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson and Carl D. Offner. *Extending OpenMP for NUMA Machines.* In SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, Dallas, Texas, USA, 2000. 36, 40, 180

[Bolosky 1993] William J. Bolosky and Michael L. Scott. *False sharing and its effect on shared memory performance.* In USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association. 29

[Brice Goglin 2009] Brice Goglin and Nathalie Furmento. *Enabling High-Performance Memory Migration for Multithreaded Applications on Linux.* In IEEE, editeur, MTAAP'09: Workshop on Multithreaded Architectures and Applications, IPDPS, Rome Italie, 2009. 42, 43, 45, 86, 180

[Broquedis 2009] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst and Pierre-André Wacrenier. *Dynamic Task and Data Placement over NUMA Architectures: an OpenMP Runtime Perspective.* In 5th International Workshop on OpenMP, pages 79–92, Dresden, Germany, 2009. Springer. 30, 38

[Broquedis 2010a] François Broquedis. *De l'exécution d'applications scientifiques OpenMP sur architectures hiérarchiques.* PhD thesis, Université Bordeaux 1, December 2010. 38, 45, 180

[Broquedis 2010b] Francois Broquedis, Jerome Clet-Ortega, Stephanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault and Raymond Namyst. *hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications.* Parallel, Distributed, and Network-Based Processing, Euromicro Conference on, pages 180–186, 2010. 38, 44

[Brunner 2000] Robert K. Brunner and Laxmikant V. Kalé. *Handling Application-Induced Load Imbalance using Parallel Objects.* In Parallel and Distributed Computing for Symbolic and Irregular Applications, pages 167–181. World Scientific Publishing, 2000. 100

[Carissimi 1999] Alexandre Carissimi. *Athapascan-0 : Exploitation de la multiprogrammation légere sur grappes de multiprocesseurs.* PhD thesis, Institut National Polytechnique de Grenoble, 1999. 9, 178

[Carissimi 2007] Alexandre Carissimi, Fabrice Dupros, Jean-François Mehaut and Rafael Vanoni Polanczyk. *Aspectos de Programação Paralela em arquiteturas NUMA.* In VIII Workshop em Sistemas Computacionais de Alto Desempenho, 2007. 43, 96

[Carlson 1999] William W. Carlson, Jesse M. Draper and David E. Culler. *Introduction to UPC and Language Specification.* Rapport technique CCS-TR-99-157, George Mason University, 1999. 25, 26

[Castro 2009a] Márcio Castro. NUMA-ICTM: A Parallel Version of ICTM Exploiting Memory Placement Strategies for NUMA Machines. Master's thesis,

Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, Brazil, 2009. 138, 140, 190, 193

[Castro 2009b] Márcio Castro, Luiz Gustavo Fernandes, Christiane Pousa Ribeiro, Jean-François Méhaut and Marilton S. de Aguiar. *NUMA-ICTM: A Parallel Version of ICTM Exploiting Memory Placement Strategies for NUMA Machines*. PDSEC '09: Parallel and Distributed Processing Symposium, International, pages 1–8, 2009. 134, 138, 140, 188, 191, 193

[Coarfa 2005] Cristian Coarfa, Yuri Dotsenko, John M. Crummey, François Cantonnet, Tarek E. Ghazawi, Ashrujit Mohanti, Yiyi Yao and Daniel C. Miranda. *An evaluation of global address space languages: co-array fortran and unified parallel C*. In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '05, pages 36–47. ACM, 2005. 25, 40

[Cray 1993] Cray. *CRAY T3D System Architecture Overview Manual*. ftp://ftp.cray.com/product-info/mpp/T3D_Architecture_Over/T3D.overview.html, 1993. 12

[Cruz 2010] Eduardo H.M. Cruz, Marco A.Z. Alves and Philippe O.A. Navaux. *Process Mapping Based on Memory Access Traces*. Computing Systems (WSCAD-SCC), 2010 11th Symposium on, 2010. 8, 127, 131

[Cruz 2011] Eduardo H.M. Cruz, Marco A.Z. Alves, Christiane P. Ribeiro, Alexandre Carissimi, Philippe O.A. Navaux and Jean-François Méhaut. *Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms*. In 13th Workshop on Advances in Parallel and Distributed Computational Models. IEEE Press, 2011. 87, 127

[Dahlgren 1999] Fredrik Dahlgren and Josep Torrellas. *Cache-Only Memory Architectures*. Computer, vol. 32, pages 72–79, 1999. 10

[Danjean 2003] Vincent Danjean and Raymond Namyst. *Controlling Kernel Scheduling from User Space: An Approach to Enhancing Applications Reactivity to I/O Events*. In Timothy Mark Pinkston and Viktor K. Prasanna, editeurs, High Performance Computing – HiPC 2003, volume 2913 of *Lecture Notes in Computer Science*, pages 490–499. Springer Berlin, 2003. 38

[de Aguiar 2004] Marilton S. de Aguiar, Graçaliz Pereira Dimuro, Antônio Carlos da Rocha Costa, Rafael K. S. Silva, Fábia A. da Costa and Vladik Kreinovich. *The Multi-layered Interval Categorizer Tesselation-based Model*. In VI Brazilian Symposium on Geoinformatics, 22-24 November, Campos do Jordão, São Paulo, Brazil, pages 437–454. INPE, 2004. 138, 190

[Dempsey 2010] Jim Dempsey. *QuickThread**. http://software.intel.com/en-us/articles/quickthread/, 2010. 39

[Dooley 2010] Isaac Dooley, Chao Mei, Jonathan Lifflander and Laxmikant Kalé. *A Study of Memory-Aware Scheduling in Message Driven Parallel Programs*. In Proceedings of 17th Annual International Conference on High Performance Computing, 2010. 39

[Dupros 2008] Fabrice Dupros, Hideo Aochi, Ariane Ducellier, Dimitri Komatitsch and Jean Roman. *Exploiting Intensive Multithreading for the Efficient Simulation of 3D Seismic Wave Propagation.* In CSE '08: Proceedings of the 11th International Conference on Computational Science and Engineerin, pages 253–260, Sao Paulo, Brazil, 2008. 134, 188

[Dupros 2009] Fabrice Dupros, Christiane Pousa Ribeiro, Alexandre Carissimi and Jean-François Méhaut. *Parallel Simulations of Seismic Wave Propagation on NUMA Architectures.* In ParCo'09: International Conference on Parallel Computing, Lyon, France, 2009. 11, 134, 135, 188, 189

[Dupros 2011] Fabrice Dupros, Christiane Pousa Ribeiro, Hideo Aochi, Jean-François Méhaut, Dimitri Komatitsch and Jean Roman. *Efficient Stencil Computation on Multicore and Hierarchical Architectures: Application to Seismic Wave Propagation.* Concurrency and Computation: Practice and Experience (submitted), 2011. 135

[Falsafi 1997] Babak Falsafi and David A. Wood. *Reactive NUMA: a design for unifying S-COMA and CC-NUMA.* In ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture, pages 229–240, New York, NY, USA, 1997. ACM. 11, 12

[Felber 2008] Pascal Felber, Christof Fetzer and Torvald Riegel. *Dynamic performance tuning of word-based software transactional memory.* In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 237–246, New York, NY, USA, 2008. ACM. 151

[Freitas 2009] Henrique C. Freitas, Marco A. Z. Alvez and Philippe O. A. Navaux. Erad 2009 - escola regional de alto desempenho multi-core, chapitre NoC e NUCA: Conceitos e Tendências para Arquiteturas Many-Core, pages 5–37. Biblioteca do Instituto de InformÃ¡tica da UFRGS, Porto Alegre, 2009. 10

[Ghemawat 2011] Sanjay Ghemawat and Paul Menage. *TCMalloc : Thread-Caching Malloc.* `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, 2011. 42, 59

[Gloger 2011] Wolfram Gloger. *The ptmalloc.* `http://www.malloc.de/en/`, 2011. 42, 59

[Góes 2010a] Luís Fabrício Góes, Marcelo Cintra and Murray Cole. *OpenSkel: Worklist Transactional Skeleton Framework.* Research Report, University of Edinburgh, 2010. 26, 53, 106, 107, 151

[Góes 2010b] Luís Fabrício Góes, Marcelo Cintra and Murray Cole. *Transactional Skeletons: Improving Performance of STM Applications using Software Helper Threads.* In Scottish Informatics and Computer Science Alliance PhD Conference, Edinburgh, 2010. SICSA2010. 24, 26, 76, 89, 106, 151, 184

[Góes 2011] Luís Fabrício Góes, Christiane Pousa, Márcio Castro, Jean-François Méhaut, Marcelo Cintra and Murray Cole. *Exploiting Memory and Cache*

Affinity in Transactional Memory Applications. In International Workshop on Languages and Compilers for Parallel Computing - LCPC (submitted), France, 2011. Springer-Verlag. 71, 151

[Grama 2003] Ananth Grama, George Karypis, Vipin Kumar and Anshul Gupta. Introduction to parallel computing. Addison-Wesley, 2003. 24

[Hackenberg 2009] Daniel Hackenberg, Daniel Molka and Wolfgang E. Nagel. *Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems*. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pages 413–422, New York, NY, USA, 2009. ACM. 20

[Huang 2003] Chao Huang, Orion Lawlor and Laxmikant V. Kalé. *Adaptive MPI*. In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958, pages 306–322, College Station, Texas, October 2003. 97

[Huang 2007] Chao Huang, Gengbin Zheng and Laxmikant V. Kalé. *Supporting Adaptivity in MPI for Dynamic Parallel Applications*. Rapport technique 07-08, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007. 95, 97

[Intel-PTU 2010] Intel-PTU. *Intel Performance-Tuning Utility*. http://www.intel.com/technology/itj/2007/v11i4/2-parallelization/1-abstract.htm, 2010. 117

[Intel-vtune 2010] Intel-vtune. *Intel VTune Performance Analyzer*. http://software.intel.com/en-us/intel-vtune/, 2010. 117

[Intel 2011a] Intel. *An Introduction to the Intel QuickPath Interconnect*. www.intel.com/technology/quickpath/introduction.pdf, 2011. 17, 18

[Intel 2011b] Intel. *Laptop, Notebook, Desktop, Server and Embedded Processor Technology - Intel*. http://www.intel.com, 2011. 17

[Iyer 1998] Ravi Iyer, Hu Wang and Laxmi Bhuyan. *Design and Analysis of Static Memory Management Policies for CC-NUMA Multiprocessors*. Rapport technique, College Station, TX, USA, 1998. 62

[Jacobi 2011] Charm++ Jacobi. *Jacobi 2D Implemented with Charm++*. http://charm.cs.uiuc.edu/tutorial/Basic2DJacobi.htm, 2011. 30, 146

[Jeremiassen 1995] Tor E. Jeremiassen and Susan J. Eggers. *Reducing false sharing on shared memory multiprocessors through compile time data transformations*. SIGPLAN Not., vol. 30, pages 179–188, August 1995. 30

[Jin 1999] Haoqiang Jin, Michael Frumkin and Jerry Yan. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*. Rapport technique 99-011/1999, NAS System Division - NASA Ames Research Center, 1999. 122, 125

[Joseph 2006] Antony Joseph, Janes Pete and Rendell Alistair. *Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, Ul-*

traSPARC/FirePlane and Opteron/HyperTransport. In High Performance Computing - HiPC, pages 338–352. 2006. 31, 43, 96, 181

[Kalé 1993] Laxmikant V. Kalé and S. Krishnan. *CHARM++: A Portable Concurrent Object Oriented System Based on C++*. In A. Paepcke, editeur, Proceedings of OOPSLA'93, pages 91–108. ACM Press, September 1993. 23, 27, 53

[Kalé 1996] Laxmikant V. Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan and Joshua Yelon. *Converse: An Interoperable Framework for Parallel Programming*. Parallel Processing Symposium, International, vol. 0, page 212, 1996. 99

[Kalé 2009a] Laxmikant V. Kalé and Gengbin Zheng. *Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects*. In M. Parashar, editeur, Advanced Computational Infrastructures for Parallel and Distributed Applications, pages 265–282. Wiley-Interscience, 2009. 27, 89, 184

[Kalé 2009b] Laxmikant V. Kalé and Gengbin Zheng. *Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects*. In M. Parashar, editeur, In Advanced Computational Infrastructures for Parallel and Distributed Applications, pages 265–282. Wiley-Interscience, 2009. 76, 95

[Kaminski 2009] Patryk Kaminski. *NUMA aware heap memory manager*. `http://developer.amd.com/Assets/NUMA_aware_heap_memory_manager\_article_final.pdf`, 2009. 42

[Karypis 1995] George Karypis and Vipin Kumar. *METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*. Rapport technique, 1995. 146, 196

[Kim 2002] Changkyu Kim, Doug Burger and Stephen W. Keckler. *An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches*. SIGOPS Oper. Syst. Rev., vol. 36, pages 211–222, October 2002. 10

[Kleen 2005] Andi Kleen. *A NUMA API for Linux*. Rapport technique Novell-4621437, 2005. 42, 43, 44, 116, 137, 180, 181

[Koenig 2007] Gregory Allen Koenig and Laxmikant V. Kalé. *Optimizing Distributed Application Performance Using Dynamic Grid Topology-Aware Load Balancing*. In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, pages 1–10, 2007. 39

[Larus 2006] James R. Larus and Ravi Rajwar. Transactional memory. Morgan & Claypool Publishers, 2006. 24

[Lenoski 1993] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta and J. Hennessy. *The DASH Prototype: Logic Overhead and Performance*. IEEE Trans. Parallel Distrib. Syst., vol. 4, no. 1, pages 41–61, 1993. v, 11, 12, 16

[Leung 2004] Joseph Y-T. Leung. Handbook of scheduling: algorithms, models, and performance analysis. Chapman & Hall/CRC computer and information science series. Chapman & Hall/CRC, 2004. 65

[Liu 2009] Mengxiao Liu, Weixing Ji, Zuo Wang and Xing Pu. *A Memory Access Scheduling Method for Multi-core Processor*. Computer Science and Engineering, International Workshop on, vol. 1, pages 367–371, 2009. 15, 178

[LMbench 2010] LMbench. *LMbench benchmark*. http://www.gelato.unsw.edu.au/IA64wiki/lmbench3, 2010. 74

[Löf 2005] Henrik Löf and Sverker Holmgren. *Affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System*. In ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing, pages 387–392, New York, NY, USA, 2005. ACM. 43

[MacDonald 2000] Steve MacDonald, Duane Szafron, Jonathan Schaeffer and Steven Bromling. *Generating Parallel Program Frameworks from Parallel Design Patterns*. In Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing, pages 95–104. Springer-Verlag, 2000. 106

[Marathe 2006] Jaydeep Marathe and Frank Mueller. *Hardware Profile-Guided Automatic Page Placement for ccNUMA Systems*. In PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 90–99, New York, NY, USA, 2006. ACM. 41

[Marr 2002] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller and Michael Upton. *Hyper-Threading Technology Architecture and Microarchitecture*. http://www.malloc.de/en/, 2002. 13

[Mccalpin 1995] John D. Mccalpin. *STREAM: Sustainable memory bandwidth in high performance computers*. http://www.cs.virginia.edu/stream/, 1995. 115, 122, 188

[McCalpin 2007] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Rapport technique, University of Virginia, Charlottesville, Virginia, 1991-2007. 36, 74

[McKee 2004] Sally A. McKee. *Reflections on the memory wall*. In Proceedings of the 1st conference on Computing frontiers, CF '04, pages 162–, New York, NY, USA, 2004. ACM. 11

[Mei 2010] Chao Mei, Gengbin Zheng, Filippo Gioachin and Laxmikant V. Kalé. *Optimizing a parallel runtime system for multicore clusters: a case study*. In TG '10: Proceedings of the 2010 TeraGrid Conference, pages 1–8, New York, NY, USA, 2010. ACM. 8, 95, 143, 194

[Minh 2008] Chi C. Minh, Jaewoong Chung, C. Kozyrakis and K. Olukotun. *STAMP: Stanford Transactional Applications for Multi-Processing*. In IISWC '08: Proceedings of the IEEE International Symposium on Workload Characterization, pages 35–46, Seattle, WA, USA, 2008. 34, 150

[Molka 2009] Daniel Molka, Daniel Hackenberg, Robert Schone and Matthias S. Muller. *Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System*. In 18th International Conference on Paral-

lel Architectures and Compilation Techniques, pages 261–270, USA, 2009. IEEE. 18, 20

[Moore 2000] Gordon E. Moore. *Readings in computer architecture.* chapitre Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. 13

[Namyst 1995] Raymond Namyst and Jean-François Méhaut. *Marcel : Une bibliothéque de processus légers.* LIFL, Univ. Sciences et Techn. Lille, 1995. 38

[Namyst 1997] Raymond Namyst. *PM2: un environnement pour une conception portable et une exécution efficace des applications paralléles irrégulieres.* PhD thesis, Université de Lille 1, 1997. 23

[Nikolopoulos 2001] Dimitrios S. Nikolopoulos, Ernest Artiaga, Eduard Ayguadé and Jesús Labarta. *Exploiting Memory Affinity in OpenMP Through Schedule Reuse.* SIGARCH Computer Architecture News, vol. 29, no. 5, pages 49–55, 2001. 36, 38, 179

[Nikolopoulos 2002] Dimitrios S. Nikolopoulos, Eduard Ayguadé and Constantine D. Polychronopoulos. *Runtime vs. Manual Data Distribution for Architecture-Agnostic Shared-Memory Programming Models.* Int. J. Parallel Program., vol. 30, no. 4, pages 225–255, 2002. 36

[NVIDIA 2010] NVIDIA. *What is Cuda?* `http://www.nvidia.com/object/what_is_cuda_new.html`, 2010. 24

[Nyland 1996] Lars S. Nyland, Jan Prins, Allen Goldberg, Peter Mills, John H. Reif and Robert A. Wagner. *A Refinement Methodology for Developing Data-Parallel Applications.* In Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume I, Euro-Par '96, pages 145–150, London, UK, 1996. Springer-Verlag. 53

[Omni Project 2010] Omni Project. *OpenMP version of the NAS Parallel Benchmarks.* `http://www.hpcs.cs.tsukuba.ac.jp/omniopenmp/download/download-benchmarks.html`, 2010. 125

[OpenMP 2011] OpenMP. *The OpenMP API Specification for Parallel Programming.* `http://www.openmp.org`, 2011. 23, 24, 25, 53, 76, 89, 184

[Papamarcos 1984] Mark S. Papamarcos and Janak H. Patel. *A low-overhead coherence solution for multiprocessors with private cache memories.* In Proceedings of the 11th annual international symposium on Computer architecture, ISCA '84, pages 348–354, New York, NY, USA, 1984. ACM. 18, 19

[PAPI 2010] PAPI. *Performance Application Programming Interface.* `http://icl.cs.utk.edu/papi/`, 2010. 117

[Patterson 2009] David A. Patterson and John L. Hennessy. Computer organization and design (4nd ed.): the hardware/software interface. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009. 7, 13, 177

[Pilla 2011a] Laércio Pilla, Christiane Pousa Ribeiro, Daniel Cordeiro, Philippe O. A. Navaux and Jean-François Méhaut. *Load Balancing for NUMA plat-*

forms. In 9th Workshop on Charm++ and its Applications, Urbana, USA, 2011. 70, 100, 186, 199

[Pilla 2011b] Laércio Pilla, Christiane Pousa Ribeiro, Daniel Cordeiro, Philippe O. A. Navaux, Jean-François Méhaut, Abhinav Bhathele and Laxmikant V. Kalé. *Improving Parallel System Performance with a NUMA-aware Load Balancer*. In International Conference on high performance computing - HiPC, HiPC'11 (submitted), India, 2011. IEEE. 100

[Pousa Ribeiro 2009] Christiane Pousa Ribeiro and Jean-François Méhaut. *Minas Project - Memory affInity maNAgement System*. `http://pousa.christiane.googlepages.com/Minas`, 2009. 180

[PPL-Charm++ 2011] PPL-Charm++. *Charm++ Parallel Programming System*. `http://charm.cs.uiuc.edu/`, 2011. vi, 23, 27, 96, 143, 146, 194, 196

[Ramamoorthy 1977] C. V. Ramamoorthy and H. F. Li. *Pipeline Architecture*. ACM Comput. Surv., vol. 9, pages 61–102, March 1977. 13

[Ribeiro 2008] Christiane Pousa Ribeiro, Fabrice Dupros, Alexandre Carissimi, Vania Marangozova-Martin and Jean-François Méhaut. *Explorando Afinidade de Memória em Arquiteturas NUMA*. In WSCAD'08: Proceedings of the 9th Workshop em Sistemas Computacionais de Alto Desempenho - SBAC-PAD, Campo Grande, Brazil, 2008. SBC. 65

[Ribeiro 2009a] Christiane Pousa Ribeiro, Márcio Castro, Luiz Gustavo Fernandes, Alexandre Carissimi and Jean-François Méhaut. *Memory Affinity for Hierarchical Shared Memory Multiprocessors*. In 21st International Symposium on Computer Architecture and High Performance Computing, São Paulo, Brazil, 2009. IEEE. 71, 134, 182, 188

[Ribeiro 2009b] Christiane Pousa Ribeiro, Márcio Castro, Luiz Gustavo Fernandes, Fabrice Dupros, Alexandre Carissimi and Jean-François Méhaut. *High Performance Applications on Hierarchical Shared Memory Multiprocessors*. In Colloque d'Informatique: Brésil / INRIA, Coopérations, Avancées et Défis, Brazil, 2009. SBC. 137, 138, 191

[Ribeiro 2009c] Christiane Pousa Ribeiro and Jean-François Méhaut. *Minas: Memory Affinity Management Framework*. Research Report RR-7051, INRIA, 2009. 137

[Ribeiro 2010a] Christiane Pousa Ribeiro, Alexandre Carissimi and Jean-François Méhaut. *Memory Access Characterization of OpenMP Workloads on a Multicore NUMA Machine*. Research Report RR-7051, INRIA, 2010. 65

[Ribeiro 2010b] Christiane Pousa Ribeiro, Alexandre Carissmi and Jean-François Méhaut. *Memory Affinity Management for Numerical Scientific Applications over Multi-core Multiprocessors with Hierarchical Memory*. In PhD Forum of 24th IEEE International Parallel and Distributed Processing Symposium, US, 2010. IEEE. 69, 70, 135, 138, 189, 191

[Ribeiro 2010c] Christiane Pousa Ribeiro, Márcio Castro, Alexandre Carissimi and Jean-François Méhaut. *Improving Memory Affinity of Geophysics Applications on NUMA platforms Using Minas*. In 9th International Meeting High Performance Computing for Computational Science, VECPAR, US, 2010. LNCS. 71, 134, 188

[Ribeiro 2010d] Christiane Pousa Ribeiro, Maxime Martinasso and Jean-François Méhaut. *NUMA Support for the charm++ Environment*. 2010. 69, 71, 97, 103, 146, 185, 196

[Ribeiro 2010e] Christiane Pousa Ribeiro, Ismael Stangherlini, Nicolas Maillard and Jean-François Méhaut. *Compiling OpenMP Applications to Enhance Memory Affinity on Hierarchical Multi-Core Machines*. In 23rd International Workshop on Languages and Compilers for Parallel Computing, US, 2010. LNCS. 71

[Richardson 1996] H. Richardson. *High Performance Fortran: history, overview and current developments*. Rapport technique TMC-261, 1996. 24, 25, 40, 180

[SGI 2011] SGI. *SGI NUMAlink Interconnect Fabric*. http://www.sgi.com/products/servers/altix/numalink.html, 2011. 17

[Singh 1993] Jaswinder Pal Singh, Truman Joe, Anoop Gupta and John L. Hennessy. *An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors*. In Supercomputing '93. Proceedings, pages 214 – 225. IEEE, 1993. 12

[Snir 2010] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra. MPI-The Complete Reference: The MPI Core. MIT Press Cambridge, 2010. 23

[Stangherlini 2010] Ismael Stangherlini. CUIA: Uma Ferramenta para a Obtenção de Informações de Variáveis em Códigos C. Master's thesis, Universidade Federal do Rio Grande do Sul, 2010. 71, 183

[Technion 2011] Technion. *Technion - Israel Institut of Technology – Advanced Cache Topics*. http://webee.technion.ac.il/courses/044800/lectures/MESI.pdf, 2011. v, 18

[Terboven 2008] Christian Terboven, Dieter A. Mey, Dirk Schmidl, Henry Jin and Thomas Reichstein. *Data and Thread Affinity in OpenMP Programs*. In MAW '08: Proceedings of the 2008 workshop on Memory access on future processors, pages 377–384. ACM, 2008. 43

[The BenchIT Project 2010] The BenchIT Project. *Performance Measurement for Scientific Applications*. http://www.benchit.org/, 2010. 115

[Thibault 2007] Samuel Thibault. *Ordonnancement de processus légers sur architectures multiprocesseurs hiérarchiques : BubbleSched, une approche exploitant la structure du parallélisme des applications*. PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, December 2007. 128 pages. 38

[Tikir 2004] Mustafa M. Tikir and Jeffrey K. Hollingsworth. *Using Hardware Counters to Automatically Improve Memory Performance*. In Proceedings of the 2004 ACM/IEEE conference on Supercomputing, SC '04, Washington, DC, USA, 2004. IEEE Computer Society. 41, 180

[UPC 2011] UPC. *Unified Parallel C*. `http://upc.gwu.edu/`, 2011. 24

[VIRTUTECH 2007] VIRTUTECH. Simics 3.0 – user guide for unix. 2007. <http://www.simics.net>. 42, 131

[Wang 2009] Zheng Wang and Michael F.P. O'Boyle. *Mapping parallelism to multicores: a machine learning based approach*. In Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP '09, pages 75–84, New York, NY, USA, 2009. ACM. 36, 39, 180

[Wentzlaff 2007] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III and Anant Agarwal. *On-Chip Interconnection Architecture of the Tile Processor*. IEEE Micro, vol. 27, pages 15–31, 2007. 16

[Wulf 1995] Wm. A. Wulf and Sally A. Mckee. *Hitting the Memory Wall: Implications of the Obvious*. Computer Architecture News, vol. 23, pages 20–24, 1995. 11

[Zheng 2005] Gengbin Zheng. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005. 100, 186

[Zheng 2006] Gengbin Zheng, Orion Sky Lawlor and Laxmikant V. Kalé. *Multiple flows of control in migratable parallel programs*. In In Proceedings of 8th Workshop on High Performance Scientific and Engineering Computing (HPSEC-06. IEEE Press, 2006. 97

Appendix A

MAi Interface

A.1 MAi-array

```
/*Functions to initialize and finalize MAi*/
void mai_init(char *filename);
void mai_final();

/*Alloc arrays - function prototype */
void* mai_alloc_1D(int nx,size_t size_item);
void* mai_alloc_2D(int nx,int ny,size_t size_item);
void* mai_alloc_3D(int nx,int ny,int nz,size_t size_item);
void* mai_alloc_4D(int nx,int ny,int nz,size_t size_item);

/*Free allocated arrays*/
void mai_free_array(void *p);

/*mai_alloc statistics*/
mai_stats mai_allocinfo();


/*Cyclic memory policy - function prototypes*/
void mai_cyclic(void *p);
void mai_skew_mapp(void *p);
void mai_prime_mapp(void *p,int prime);

/*Bind memory policy - function prototypes*/
void mai_bind_all(void *p);
void mai_bind_all_mynode(void *p);
void mai_bind_rows(void *p);
void mai_bind_columns(void *p);
void mai_bind_rows_mynode(void *p);
void mai_bind_columns_mynode(void *p);

/*Next-touch - function prototype*/
void mai_next_touch(void *p);

/*Migration - function prototypes*/
void mai_migrate(void *p,unsigned long node,int nr, int startr,
                 int nc, int startc);
void mai_migrate_rows(void *p,unsigned long node,int nr, int start);
void mai_migrate_columns(void *p,unsigned long node,int nc, int start);
void mai_migrate_scatter(void *p,unsigned long nodes);
void mai_migrate_gather(void *p,unsigned long node);

/*Compute array blocks - function prototype*/
```

```
void mai_subarray(void *p,int dim[]);
void mai_bytes(void* p,size_t bytes);

/*data distribution - function prototype */
int mai_regularbind(void *p);
int mai_irregularbind(void *p,size_t bytes,int node);

/*nodes settings - function prototype*/
void mai_nodes(unsigned long nnodes, int *nodes);
float mai_nodebandwidth(int node);
float mai_numafactor(int node1, int node2);
float mai_bandwidth(int node1, int node2);

/*Page information - function prototype*/
void mai_print_pagenodes(unsigned long *pageaddrs,int size);
int mai_number_page_migration(unsigned long *pageaddrs,int size);
double mai_get_time_pmigration();

/*Thread information - function prototype*/
void mai_print_threadcpus();
int mai_number_thread_migration(unsigned int *threads,int size);
double mai_get_time_tmigration();

/*Memory placement information*/
void mai_get_log();
void mai_mempol(void* ph);
void mai_arraynodes(void* ph);
```

A.2 MAi-heap

```
/*Functions to initialize and finalize MAi*/
void mai_init(int MEMPOL);
void mai_final();

/*Functions to allocate data*/
void* mai_malloc(size_t size);
void* mai_malloc_local(size_t size);

/*Memory Policies for a Heap*/
int mai_bind_local(mai_heap_t *h,int nid);
int mai_bind_all(mai_heap_t *h,int nid);
int mai_cyclic_neighbor(mai_heap_t *h,int nid);
int mai_cyclic(mai_heap_t *h);
int mai_skew_mapp(mai_heap_t *h);
int mai_prime_mapp(mai_heap_t *h, int prime);

/*Memory Policies for a memory range*/
int mai_cyclic(void *ptr, size_t size, int *nodemask, int nnodes);
int mai_cyclic_block(void *ptr,size_t size, int *nodemask, int nnodes);
int mai_bind(void *ptr, size_t size, int node);
int mai_bind_block(void *ptr, size_t size,int *nodemask, int nnodes);
int mai_prime(void *ptr, size_t size, unsigned long *nodemask,
              int nnodes, int prime);
```

```
int mai_prime_block(void *ptr, size_t size, int *nodemask,
                    int nnodes, int prime);
int mai_skew(void *ptr, size_t size, int *nodemask, int nnodes);
int mai_skew_block(void *ptr, size_t size, int *nodemask, int nnodes);
int mai_random(void *ptr, size_t size, int *nodemask, int nnodes);
int mai_random_block(void *ptr, size_t size, int *nodemask, int nnodes);
int mai_next_touch(void *ptr,size_t size);

/*statistics of the memory allocator*/
void mai_HeapInfo();
int print_mem_affinity();
```

Appendix B

Extended Abstract in French

B.1 Introduction

Aujourd'hui, de nombreuses applications de calcul provenant de différents domaines scientifiques (géophysique, climatologie, physique des matériaux, etc) exigent des temps de réponse faible et de grandes capacités en mémoire. Les entreprises et organisations développant et utilisant ces applications se doivent d'acquérir des plates-formes de calcul à haute performance (HPC [1]). Un type de plate-forme qui rencontre un grand succès auprès des entreprises est le serveur de calcul multiprocesseurs à mémoire partagée (SMP [2]). Avec l'avènement des technologies de processeurs multi-cœurs, les serveurs de calcul sont devenus des machines multiprocesseurs et multi-cœurs. Les différents cœurs des serveurs se partagent les accès à la mémoire globale et disposent de plusieurs niveaux de cache pour réduire la latence mémoire.

La tendance actuelle est de voir le nombre de cœurs dans les processeurs augmenter. Les serveurs de calcul SMP disposent d'un nombre de cœurs de plus en plus important. Tous ces cœurs sont en compétition pour l'accès aux données stockées en mémoire. La contention mémoire constitue un des problèmes à appréhender sur ce type d'architecture. Pour faire face à ces problèmes de contention mémoire, les serveurs multiprocesseurs sont dotés d'un espace hiérarchique de mémoire de type NUMA [3]. Chaque processeur ou groupe de processeurs dispose d'un banc de mémoire. La mémoire du système est constituée de l'ensemble des bancs mémoires associés à chaque processeur ou groupe de processeur. Un processeur peut accéder, soit à des données stockées sur son banc local de mémoire, soit à un banc de mémoire distant. Une des problématiques associées à cette hiérarchie mémoire consiste à exploiter au mieux cet espace distribué de mémoire (répartir les données sur les bancs mémoire) et aussi à réduire les temps d'accès à ces données.

Pour obtenir de bonnes performances sur ces architectures multiprocesseurs à mémoire hiérarchique, il est indispensable de déterminer un placement judicieux des threads sur les processeurs et cœurs disponibles, mais également un placement des données des applications sur les différents bancs mémoire. Il s'agit donc de pouvoir minimiser la latence d'accès aux données et de maximiser l'utilisation de la bande passante mémoire. Cela constitue le sujet central de cette thèse.

Les constructeurs informatiques fournissent des systèmes d'exploitation et des

1. High Performance Computing
2. Symetric MultiProcessing
3. Non Uniform Memory

environnements de programmation ne permettant d'exploiter qu'une partie du potentiel de performance de leurs plates-formes. Aucune solution n'est proposée aux développeurs pour optimiser le placement efficace des threads et des données sur la hiérarchie mémoire. Les développeurs doivent implémenter leurs propres stratégies de placement. Ces stratégies sont souvent fortement liées aux caractéristiques des plates-formes. Un des objectifs est également la portabilité des performances, c'est à dire à que les stratégies de placement puissent s'adapter aux caractéristiques des plates-formes et de leurs hiérarchies mémoire.

B.2 Objectifs et contributions de la thèse

En raison des problèmes décrits précédemment, il est important de comprendre l'impact de l'affinité mémoire sur les applications parallèles. Ainsi, l'étude des solutions de l'affinité mémoire de l'état de l'art pour ces plates-formes NUMA est important pour identifier les problèmes de telles solutions. On remarquera au fil des chapitres que la plupart des outils disponibles pour contrôler l'affinité mémoire n'ont pas été standardisés et ne sont pas disponibles sur les plates-formes actuelles. C'est pourquoi ces solutions demandent un développement complexe en transformant le code source pour prendre en compte des caractéristiques de la plate-forme NUMA. En plus, les environnements de programmation ne fournissent pas les mécanismes pour contrôler l'affinité mémoire.

Comme les applications parallèles ont des caractéristiques et des besoins différents, un placement efficace des données doit être réalisé afin d'optimiser l'affinité mémoire. Par conséquent, il faut impérativement adapter la solution pour qu'elle corresponde à la machine et à ses caractéristiques, ainsi qu'aux caractéristiques de l'application. Nous proposons un environnement de programmation pour gérer l'affinité mémoire sur des plate-formes multi-cœur avec un espace mémoire NUMA. Cet environnement de programmation fournit le support nécessaire pour exprimer l'affinité des données à différents niveaux (par exemple pour les variables globales ou les variables allouées dans le tas). Lors de l'exécution de l'application, cet environnement interpole les informations sur le comportement de l'application et les informations sur l'architecture. De cette façon, les développeurs peuvent produire des applications qui prennent en compte les accès mémoire et les caractéristiques NUMA des plates-formes.

Les principales contributions de cette thèse sont: (i) un modèle qui définit et caractérise une plate-forme NUMA; (ii) la conception et la mise en œuvre d'un environnement de programmation pour gérer l'affinité mémoire des applications nommé *Minas*; (iii) l'intégration des mécanismes dans différentes applications parallèles (basées sur OpenMP, Charm++, AMPI et *OpenSkel*) et les systèmes de programmation parallèle (Charm++ et *OpenSkel*).

B.3 Contexte scientifique

Cette thèse a été développée dans le cadre du projet ANR NUMASIS [4] et dans le contexte du Laboratoire commun INRIA Université de l'Illinois, l'INRIA et la NCSA [5]. Ces organisations et entreprises ont collaboré pour concevoir des solutions efficaces pour le calcul haute performance.

Le contexte scientifique du projet NUMASIS est le calcul haute performance avec des applications issues du domaine de la géophysique. Considérant ce contexte, les problèmes viennent de l'optimisation d'applications qui simulent la propagation d'ondes sismiques sur les plates-formes NUMA. Les résultats obtenus dans ce projet ont contribué à BULL, au BRGM et au CEA dans la conception de nouvelles solutions pour les machines multi-cœur avec des caractéristiques NUMA.

Le laboratoire commun avec Urbana scientifique s'intéresse aux défis posés par les logiciels pour le calcul petaflopique. Les problèmes qui sont étudiés par cette collaboration sont liés à la modélisation et l'optimisation des bibliothèques numériques, à des questions de tolérance aux pannes et des nouveaux modèles de programmation. Les résultats obtenus par cette initiative ont permis aux deux pays d'améliorer le logiciel pour les plates-formes de haute performance pétaflopiques.

Le domaine de recherche de cette thèse se situe dans le contexte du Laboratoire d'Informatique de Grenoble et de l'équipe de recherche Mescal. Notre rôle est lié à l'étude de l'impact de l'affinité mémoire et à effectuer des propositions de solutions pour la gestion de l'affinité mémoire dans les applications parallèles.

B.4 État de l'art matériel

La tendance actuelle dans le calcul haute performance pour optimiser les performances est d'augmenter le nombre de cœurs par processeur disponibles sur les machines à mémoire partagée. La conception des puces multi-cœur et les efforts pour surmonter les limitations matérielles des multiprocesseurs symétriques (SMP) ont conduit à l'émergence des architectures hiérarchiques. Ces architectures sont construites autour d'une topologie complexe et d'un sous-système de mémoire hiérarchique. Dans ce chapitre, nous présentons l'état de l'art sur des architectures multi-cœurs avec mémoire partagée hiérarchique.

Nous définissons comme architecture à mémoire partagée hiérarchique, toute plate-forme multiprocesseur qui dispose: (i) d'unités de calcul qui partagent une mémoire globale (ii) et d'unités de calcul et mémoire organisées de façon hiérarchique. Dans ce contexte, des exemples de machines hiérarchiques sont: UMA (Uniform Memory Access) machines, NUMA (Non-Uniform Memory Access) machines, COMA (Cache Only Memory access machines) et NUCA (Non-Uniform Cache Access) [Patterson 2009].

4. Adaptation et optimisation des performances applicatives sur les architectures NUMA: Étude et mise en ouvre sur des applications en SISmologie - URL: http://numasis.gforge.inria.fr

5. laboratoire commun pour le calcul petaflopique - URL: http://jointlab.ncsa.illinois.edu/

Dans cette thèse, nous nous sommes particulièrement intéressés à des architectures hiérarchiques avec mémoire partagée qui présentent des coûts d'accès mémoire non uniformes: les plates-formes NUMA. Ces plates-formes présentent des topologies complexes et sous-systèmes de mémoire hiérarchique, qui doivent être bien exploités afin d'obtenir un maximum de performance applicative.

Le concept de processeur multi-cœur est une tendance forte dans les différents domaines de l'informatique et des architectures, spécialement dans le calcul haute performance (HPC). Ce concept constitue une réponse à quelques questions telles que l'exécution parallèle d'instructions sur une puce et la question de la consommation énergétique[Liu 2009]. Ces problèmes sont en partie résolus par le concept multi-cœur, car elle fournit plus des mémoires cache, les pipelines et des unités de calcul qui réduisent aussi la consommation. Toutefois, certains problèmes demeurent et sont liés aux caractéristiques des multi-cœurs. Par exemple, le problème des accès à la mémoire principale est une question importante.

Dans le contexte de plates-formes hiérarchiques à mémoire partagée, les processeurs multi-cœur ont été utilisés comme brique de base. En utilisant les processeurs multi-cœur, les ingénieurs et chercheurs des constructeurs ont conçu des architectures puissantes à mémoire partagée avec des dizaines voire des centaines de cœurs. Le problème de performance pour l'accès à des données apparaît lorsque tous ces cœurs ont besoin de données qui ne sont pas présentes dans les différents caches. Ces données sont donc stockées dans la mémoire globale partagée et l'accès à ces données peut consommer de nombreux cycles CPU en raison de potentielles connections à la mémoire globale. Pendant les cycles de CPU ces cœurs sont inactifs et en attente de données. Ces cycles d'attente des données sont une des sources principales d'inefficacité des applications scientifiques.

Une plate-forme NUMA est un système multiprocesseur dans lequel les éléments de calcul sont connectés à plusieurs bancs de mémoire, physiquement distribués. Bien que la mémoire soit physiquement distribuée, elle est perçue par les unités de calcul et le système d'exploitation comme une mémoire partagée unique. Sur ce type d'architecture mémoire, le temps passé pour l'accès aux données est conditionné par la distance entre le processeur et le banc mémoire dans lequel les données sont physiquement stockées [Carissimi 1999]. Les processeurs utilisées dans les architectures NUMA disposent généralement de plusieurs niveaux de cache, afin de réduire le temps d'accès aux données. Pour cette raison, la cohérence de cache des unités de calcul est mis en œuvre dans les plates-formes NUMA actuelles, conduisant à des plates-formes NUMA cache-cohérente (ccNUMA). Un des avantages de l'architecture NUMA est qu'elle combine une bonne évolutivité par rapport à la mémoire avec un modèle de programmation facile. Dans ces machines, un réseau d'interconnexion efficace et spécialisé permet de relier un nombre élevé de processeurs et de cœurs. Puisque la mémoire est logiquement partagée, les programmeurs peuvent utiliser des modèles de programmation à mémoire partagée à base de threads pour développer des applications parallèles.

Pour les machines multi-cœur avec une architecture NUMA, certains supports matériels sont inclus dans la machine pour manipuler le mémoire partagée et physique-

ment distribuée. Ce matériel peut être mis en œuvre comme un contrôleur de mémoire intégré à la puce des processeurs. Ce principe a été adopté par plusieurs constructeurs car il évite une gestion centralisée du traitement des demandes d'accès en mémoire [Awasthi 2010].

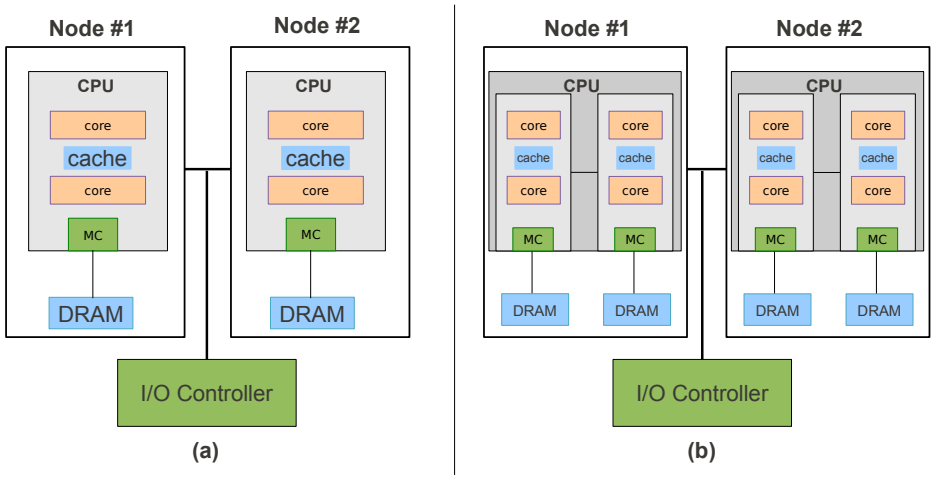


Figure B.1: Plate-forme Multi-cœur NUMA: (a) Contrôleur Mémoire Unique (b) Contrôleur Mémoire Multiple.

Les processeurs multi-cœurs, comme Nehalem d'Intel et Opteron d'AMD constituent les briques de base des plates-formes NUMA. La figure B.1 (a) présente une plate-forme NUMA avec un seul contrôleur mémoire par nœud NUMA et (b) présente une architecture NUMA avec plusieurs contrôleurs mémoire par nœud NUMA.

B.5 État de l'art logiciel

Dans ce chapitre, nous abordons les problèmes logiciels liés à la gestion de l'affinité mémoire sur des plates-formes multi-cœur NUMA. Le chapitre se poursuit par l'état de l'art au niveau des logiciels, soulignant les inconvénients du support d'affinité mémoire pour les machines actuelles. Nous présentons les approches et le support logiciel pour faire face à la gestion de l'affinité mémoire sur les machines à mémoire partagée. Enfin, nous présentons une conclusion sur les différentes solutions logicielles pour gérer l'affinité mémoire.

Ces solutions ont été conçues à différents niveaux de la pile logicielle tels que celui des bibliothèques, compilateurs, commande *shell*, support exécutif, et allocateur mémoire. Elles peuvent être classifiées en trois catégories: placement des tâches, placement des données ou un mixte des deux.

L'ordonnancement de tâches améliore l'affinité mémoire en rapprochant les tâches de leurs données et d'autres tâches avec lesquelles elles interagissent [Nikolopoulos 2001]. Avec ce type de solution, le placement peut être réalisé de façon statique, avant

l'exécution de l'application ou de façon dynamique, pendant l'exécution de l'application. Dans le placement dynamique, l'ordonnanceur peut utiliser les informations d'exécution de l'application sur la plate-forme pour placer les tâches [Wang 2009, Broquedis 2010a].

Sur les machines NUMA, les stratégies basées sur un placement des données peuvent également être utilisées afin d'améliorer l'affinité mémoire. Ces stratégies sont généralement mises en œuvre en utilisant des politiques mémoire spécialisées pour des machines NUMA. Une politique mémoire définit comment les données sont placées sur les bancs mémoire de la machine et la granularité utilisée pour ce placement. Elle vise à améliorer la localité des données pour une équipe de tâches, et dans le même temps, fournit une bande passante efficace pour accéder aux données [Richardson 1996, Benkner 2002, Bircsak 2000, Tikir 2004]. En revanche, les allocateurs de mémoire utilisent la topologie machine pour effectuer chaque allocation de données. Ce type de solution optimise la latence d'accès aux données en les allouant dans le nœde de la tâche [Kleen 2005].

L'utilisation du placement des données et tâches comme une solution pour l'affinité mémoire est basée sur une redistribution de tâches et de données en considérant les modes d'accès aux données. À notre connaissance, une seule solution a été proposée dans ce contexte et elle est décrite dans [Broquedis 2010a]. Dans ce travail, les auteurs combinent deux solutions, MAMI et ForestGOMP [Brice Goglin 2009] pour effectuer le placement des tâches et des données pour améliorer l'affinité mémoire. Cette approche exige des développeurs une connaissance de l'application afin de corriger le placement des données. Les développeurs doivent modifier le code source de l'application, afin d'informer le support executif ForestGOMP où les accès aux données ont été effectués. De cette façon, ForestGOMP et MAMI peuvent dynamiquement replacer les tâches et les données afin de réduire la latence. Cette approche exige l'intégration de nouveaux mécanismes dans le support exécutif du langage parallèle.

B.6 Minas framework

Minas [Pousa Ribeiro 2009] est un intergiciel efficace et portable qui permet aux développeurs de gérer l'affinité mémoire de manière explicite ou automatique sur les grandes plates-formes de calcul NUMA. Dans ce travail, l'efficacité est liée aux différents moyens pour contrôler l'affinité mémoire et des performances similaires sur les différentes plates-formes NUMA. Par la portabilité, nous entendons l'abstraction de l'architecture et le compilateur et des modifications minimes ou inexistantes dans le code source des applications.

Cet interlogiciel est composé de trois modules: *Minas-MAi*, *Minas-MApp* et *numarch*. *Minas-MAi* est une interface de haut niveau qui est responsable de la mise en œuvre du mécanisme explicite. *Minas-Mai* permet le contrôle de l'affinité sur les applications alors que le pré-processeur *Minas-MApp* met en œuvre un mécanisme automatique par une analyse du code source. Le dernier module, *numarch*, est quant à lui, chargé d'extraire les informations de la plate-forme cible. Ce module peut être

utilisé par le développeur pour consulter des informations sur l'architecture et il est également utilisé par *Minas-MAi* et *Minas-MApp*.

Minas se distingue des environnements concurrents par le fait qu'il se focalise sur l'optimisation de l'affinité mémoire [Joseph 2006, Kleen 2005]. Quatre aspects sont mis en avant dans *Minas*. Tout d'abord, *Minas* offre la portabilité du code source. Avec *numarch* qui fournit l'abstraction d'architecture, le développeur n'a pas besoin de spécifier les nœuds qui seront utilisés par *Minas* pour placer les données. Deuxièmement, *Minas* est flexible car il supporte deux mécanismes distincts pour contrôler l'affinité mémoire (explicite et automatique). Troisièmement, *Minas* est conçu pour des applications basées sur les tableaux. Ce choix se justifie par le fait que cette structure de données représente généralement les variables les plus importantes dans les calculs des applications du HPC. Enfin, *Minas* offre plusieurs politiques mémoire pour traiter les applications régulières (tâches accédant toujours au même ensemble de données) et les applications irrégulières (tâches accédant un ensemble non connu à la compilation de données).

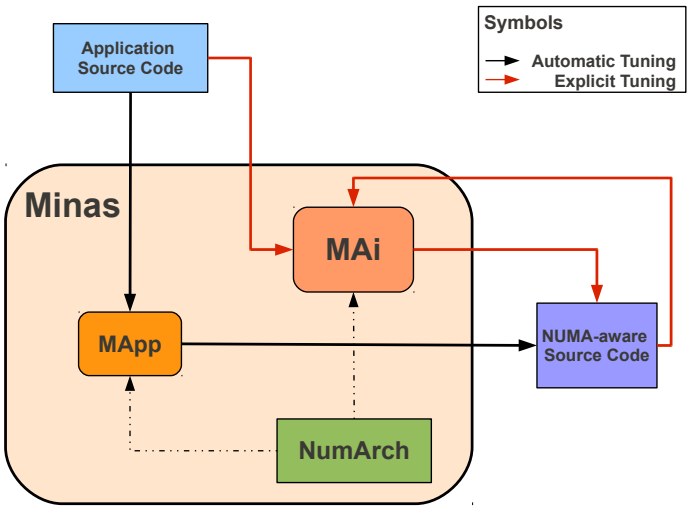


Figure B.2: Schéma du Minas.

La figure B.2 montre le schéma des approches de *Minas* pour contrôler l'affinité mémoire. Le code de l'application source d'origine peut être modifié, soit en utilisant le mécanisme explicite (flèches rouges), soit en utilisant le mécanisme automatique (flèches noires). Dans le cas du mécanisme explicite le programmeur doit modifier le code source de l'application manuellement dans le but d'améliorer l'affinité mémoire. Dans le cas du mécanisme automatique le code source des applications est automatiquement changé par *Minas*. La décision entre le mécanisme automatique et explicite dépend des connaissances du développeur sur l'application et la plateforme. Une approche possible est d'utiliser le mécanisme de *MApp* et de vérifier si les performances sont suffisantes. Si le gain n'est pas satisfaisant, les développeurs

peuvent ensuite modifier explicitement (réglage manuel) le code source des applications en utilisant *MAi*. Les composants *MAi* et *MApp* s'appuient sur *numarch* pour récupérer une partie des informations sur la machine et ses performances du sous-système mémoire.

Selon le mécanisme, *numarch* est utilisé pour rassembler des informations différentes de la machine. Pour le mécanisme explicite, *Minas-Mai* récupère de *numarch* le nombre de nœuds et de processeurs ainsi que leurs identificateurs physiques afin d'appliquer les politiques mémoire. Dans le mécanisme automatique, *MApp* reçoit de *numarch* le facteur NUMA de la machine, la bande passante d'interconnexion, l'information du sous-système de mémoire cache et la quantité de mémoire libre de chaque nœud. Ces informations sont ensuite utilisées par l'heuristique pour déterminer la politique mémoire appropriée pour les données. La politique mémoire choisie sera appliquée dans l'application en utilisant les fonctions de *MAi*.

MAi (Memory Affinity interface) est une API (Application Programming Interface) qui fournit un moyen simple de contrôler l'affinité mémoire [Ribeiro 2009a]. *MAi* simplifie la gestion des problèmes d'affinité, car elle fournit des fonctions de niveau simple et élevé qui peuvent être appelées dans le code source de l'application pour placer les données sur les bancs mémoire. Toutes les fonctions de *MAi* sont basées sur des structures de données comme les vecteurs ou les matrices ainsi que structures des données dynamiques comme les listes.

Le groupe le plus important de fonctions *MAi* est le groupe des politiques mémoire, car il est chargé d'assurer l'affinité mémoire. L'interface met en œuvre plusieurs politiques mémoire qui ont pour unité d'affinité les variables des applications. Les politiques mémoire peuvent être divisées en deux groupes: *bind* et *cyclique*. Les politiques *bind* vont optimiser la latence, en plaçant les données et les tâches aussi près que possible. Les politiques *cycliques* vont optimiser la bande passante, car elles assurent une meilleure utilisation de l'interconnexion et des bancs mémoire.

Le groupe *bind* a deux politiques mémoire, *bind_ block* et *bind_ all*. Dans le *bind_ block*, les données sont divisées en blocs selon le nombre de tâches qui seront utilisées et chaque bloc est placé sur un banc mémoire de la machine. Dans *bind_ all*, les données sont placées dans un ou plusieurs ensembles pour restreindre des nœuds. Le groupe *cyclique* est composé de *cyclic*, emph skew_mapp et de *prime_ mapp*. Dans *cyclic*, les données sont placées selon une distribution round-robin, en utilisant une page de mémoire par tour. Dans le*skew_ mapp*, une page i est allouée sur le nœud $(i+\lfloor i/M \rfloor+1)$ *mod* M, où M est le nombre de banc mémoire. Le *prime_ mapp* utilise une stratégie à deux phases. Dans la première phase, la politique utilise *cyclic* sur (P) bancs de mémoire virtuelle, où P est un nombre premier supérieur ou égal à M (nombre réel de bancs de mémoire). Dans la deuxième phase, les pages de mémoire mises sur les bancs de mémoire virtuelles sont réorganisées et placées sur des bancs de mémoire réels en utilisant le politique *cyclic*.

MAi permet aussi au développeur de modifier la politique mémoire et d'appliquer à une variable pendant l'exécution application, permettant d'exprimer des accès différents aux données. Enfin, tout placement mémoire incorrect peut être optimisé

par l'utilisation des fonctions de migration de mémoire.

MApp (Memory Affinity preprocessor) est un préprocesseur qui offre un contrôle transparent de l'affinité mémoire pour les applications scientifiques numériques du HPC sur des plate-formes NUMA. *MApp* effectue des optimisations dans le code source des applications en tenant compte des variables et les caractéristiques de la plate-forme au moment de la compilation. Ses caractéristiques principales sont sa simplicité d'utilisation (aucune modification manuelle) et s'indépendance de la plate-forme et du compilateur.

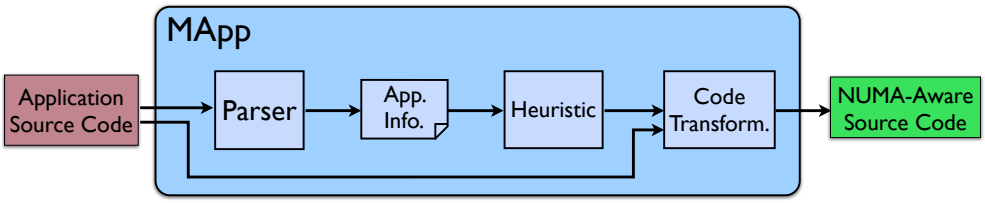


Figure B.3: MApp - Processus de transformation du code.

La figure B.3 montre le schéma du processus pour améliorer l'affinité mémoire pour les applications en utilisant *MApp*. Le processus commence par l'extraction d'information des variables des programmes. Puis, le code de l'application originale est traité par l'analyseur *MApp*, nommé CUIA (Code Under examInation to retrieve informAtion) [Stangherlini 2010]. Ensuite, il récupère les caractéristiques de la plate-forme à partir du module *numarch*. L'heuristique du module *MApp* utilise ces informations pour déterminer la politique de la mémoire pour chaque variable. Enfin, le module de transformation modifie le code source de l'application.

Le module *numarch* a un rôle important pour *Minas*, car il récupère les caractéristiques de la machine qui sont nécessaires pour placer les données sur les bancs mémoire. Ce module extrait les informations du réseau d'interconnexion (nombre de liens et la bande passante), les coûts d'accès mémoire (facteur NUMA et de latence) et les caractéristiques d e l'architecture (nombre de nœuds, les processeurs et sous-système de cache). Pour récupérer ces informations, une analyse des fichiers du système d'exploitation est effectuée. Les informations récupérées sont stockées dans des fichiers temporaires qui seront après utilisés par *numarch*.

Ce module peut également être utilisé comme une bibliothèque, car il fournit des fonctions de haut niveau qui peuvent être appelées sur le code source de l'application pour obtenir des informations sur la machine NUMA cible. La bibliothèque est composée par un ensemble de fonctions pour récupérer des informations comme le nombre de nœuds, la taille du cache, le total de mémoire libre sur chaque nœud, et le nombre de cœurs par processeur. Ces informations peuvent être utilisées par le développeur afin d'avoir une connaissance parfaite de la topologie de la machine et de ses caractéristiques.

B.7 Intégration dans les langages parallèles

Plusieurs langages parallèles sont disponibles pour programmer des plates-formes multi-cœur à mémoire partagée. Cependant, tel que présenté dans le chapitre 3, la plupart d'entre eux n'ont pas de support d'affinité mémoire. Pour cette raison, leur performance peut être limitée sur les plates-formes NUMA en raison des coûts des accès à la mémoire. Par exemple, OpenMP [OpenMP 2011], Charm++ [Kalé 2009a] et *OpenSkel* [Góes 2010b] sont des d'interfaces de programmation qui n'ont pas un support NUMA.

Dans cette section, nous montrons comment utiliser les composants *Minas* sur les interfaces parallèles pour le calcul haute performance. Pour montrer l'applicabilité des approches de *Minas* et comment elles peuvent être utilisées pour améliorer les performances des applications parallèles sur plates-formes NUMA, nous avons sélectionné deux environnements parallèle différents: OpenMP et Charm++. Nous les avons choisis parce que ils manquent de support NUMA, ils représentent des modèles de programmation différents et ils peuvent fournir des informations de l'application au moment de la compilation ou de l'exécution. Nous présentons comment utiliser les approches *Minas* pour contrôler l'affinité mémoire pour chacun des environnements choisis.

B.7.1 OpenMP

Nous proposons un support d'affinité mémoire transparent pour OpenMP, qui ne demande pas des changements explicites dans le code source de l'application, ni dans l'interface OpenMP et ni dans l'environnement d'exécution. Le support automatique d'affinité mémoire pour les applications OpenMP s'appuie sur l'idée d'utiliser les informations obtenues par le compilateur et de l'abstraction architecture pour changer le code source de l'application. Par conséquent, toutes les composantes *Minas*, *numarch*, *MAi* et *MApp* sont utilisées pour contrôler l'affinité mémoire pour les applications OpenMP. Toutefois, comme les applications OpenMP utilisent généralement des tableaux, l'interface *MAi* utilisée pour ces applications est le MAi-array.

numarch est utilisé par *MAi-array* et *MApp* afin de récupérer les informations de la machine NUMA sans l'intervention du programmeur. Cela signifie que l'abstraction de l'architecture pour les applications OpenMP est garantie par *numarch*. En utilisant les informations de l'architecture, *MAi-array* est capable de placer les tâches et les données sur les nœuds NUMA utilisant l'une des stratégies de placement des tâches et politiques de la mémoire présentées au chapitre 5.

Le choix de la politique mémoire que doit être appliquée pour une variable s'appuie sur le préprocesseur *MApp*. *MApp* analyse l'application OpenMP au moment de la compilation pour extraire des informations du code source, telles que les variables et leurs modèles d'accès mémoire. Ce préprocesseur compare et met en relation les informations des variables avec les caractéristiques de la plate-forme NUMA pour produire automatiquement un code NUMA OpenMP. *MApp* produit

ce code en insérant des fonctions de l'interface MAi-array dans le code source de l'application pour allouer et placer les données. Le choix de la politique mémoire à utiliser pour chaque tableau est fait en utilisant l'heuristique introduit dans le chapitre 4. Le code source final peut être utilisé avec n'importe quel compilateur qui dispose du support OpenMP.

Bien que *MApp* analyse l'application et les caractéristiques de la machine pour gérer l'affinité mémoire, il peut parfois produire un code source dont les performances sont en deçà de celles attendues. *MApp* est basé sur une heuristique qui ne produit pas toujours la solution optimale. De ce fait, nous permettons au programmeur d'utiliser l'interface *MAi* pour modifier manuellement le code source de son application.

B.7.2 Charm++

Le premier support que nous avons proposé pour Charm++ est l'intégration de *MAi* dans son support executif [Ribeiro 2010d]. *MAi* fournit aux programmeurs Charm++ des politiques mémoire pour placer les données de leurs applications sur une machine NUMA de façon transparente. C'est à dire que les développeurs n'ont pas besoin de modifier le code source de leurs applications. L'interface *MAi* pour Charm++ est disponible en ligne de commande, ce qui permet aux utilisateurs de sélectionner la politique mémoire à partir d'une liste des stratégies possibles pour une exécution de l'application.

Il est important de mentionner que pour Charm++, nous utilisons des politiques mémoire appliquée aux données d'une tâche (Charm++ threads). L'interface *MAi* place les données de une application sur les nœuds spécifiés par l'utilisateur suivant la politique mémoire sélectionnée. Considérant les caractéristiques de Charm++, nous employons trois politiques mémoire de *MAi*: *bind all, cyclic, cyclic neighbors*. La figure B.4 montre les différences entre les politiques mémoire; les couleurs sont utilisées pour représenter les tâches et leurs données.

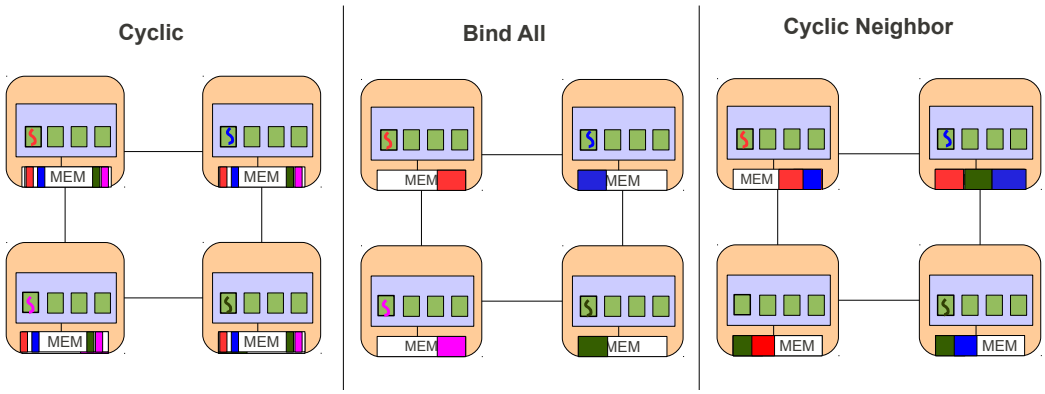


Figure B.4: Politiques mémoire pour Charm++.

La politique mémoire *bind all* place les données d'une tâche sur un banc de

mémoire physique, afin d'associer des données d'une tâche au nœud où la tâche est en train de s'exécuter. Cette politique mémoire est particulièrement adaptée pour les applications, où la tâche alloue ses propres données et les utilise exclusivement lors de l'exécution de l'application. Cette politique fonctionne de manière similaire au *first-touch*, la différence est que l'utilisateur peut sélectionner les bancs mémoire qu'une tâche doit utiliser pour allouer ses données et pas seulement celui du œud d'exécution. De cette façon, les utilisateurs peuvent par exemple exclure les bancs de mémoire attachés à des nœuds qui effectuent des opérations d'entrées-sorties (I/O) qui sont relativement sollicités et moins disposés au calcul.

Le politique mémoire *cyclic* distribue des données sur tous les nœuds NUMA d'une manière cyclique alors que le *cyclic neighbors* répartit des données dans un sous-ensemble de nœuds NUMA de la machine. Toutefois, *cyclic neighbors* va uniquement utiliser les nœuds NUMA qui sont des voisins du nœud où la tâche est en cours d'exécution. Dans le cas de Charm++, *Minas* ne place pas les tâches sur les processeurs de la machine. De plus, Charm++ dispose d'un support au placement des tâches, nous nous appuierons donc sur ce support pour placer les tâches d'une application.

Différentes approches peuvent être utilisées pour améliorer l'affinité mémoire sur des machines NUMA. Des allocateurs de mémoire dynamique, des politiques de mémoire, des mécanismes pour placer des tâches, un mécanisme d'équilibrage de charge peut également être utilisé pour améliorer l'affinité mémoire sur des machines NUMA. Le placement des tâches exige une connaissance a priori des caractéristiques de l'application (par exemple les modèles d'accès mémoire, modèle de communication) pour placer les tâches efficacement sur la machine. Dans Charm++ les mécanismes d'équilibrage n'ont pas une connaissance a priori des caractéristiques de l'application. Par conséquent, nous devons utiliser des techniques de profilage. L'utilisation d'un tel mécanisme à l'intérieur de Charm++ exige donc une analyse statique de l'application parallèle.

Cependant, le support exécutif du Charm++ fournit une interface d'équilibrage de charge qui récupère des informations significatives de l'exécution d'une application [Zheng 2005]. L'environnent d'exécution Charm++ capture les statistiques des charges lors de l'exécution, qui peuvent être utilisées pour améliorer l'équilibrage de charge et, par conséquent, pour améliorer l'affinité mémoire sur les machines multicœur NUMA. Cependant, il manque encore des informations sur les coûts des accès mémoire, ce qui représente un aspect important de la plate-forme NUMA. En utilisant l'interface de Charm++ et le module *numarch*, nous proposons un équilibrage de charge pour les machines NUMA nommé NumaLB [Pilla 2011a].

L'équilibrage de charge NumaLB repose sur l'idée qu'il faut faire correspondre les caractéristiques de l'application avec celles de la machine NUMA pour améliorer l'affinité mémoire. *NumaLB* s'appuie sur un ordonnancement de liste du type glouton, qui prend la tâche plus grosse (temps d'exécution) non attribuée et l'alloue sur le cœur le moins chargé. Le choix de l'algorithme glouton est basé sur l'idée de convergence rapide à une situation équilibrée en considérant d'abord les éléments provoquant les déséquilibres. Puisque l'objectif de *NumaLB* est de réduire les coûts

de communication, les tâches seront migrées vers les processeurs proches. Afin d'évaluer quel est le processeur le plus proche, nous proposons l'heuristique définie par l'équation suivante:

$$W(k,i) = L(i) + \alpha \times (-M(k,i) + \sum_{j!=i}^{nproc} (\text{M(k,j)} \times \text{NF(i,j)})) \tag{B.1}$$

où:
- W(k,i) est le poids de la chare k vers le cœur i
- L(i) est la charge du cœur i
- M(k,i) est le nombre des messages envoyé entre la chare k et les chares sur le cœur i
- NF(i,j) est le facteur NUMA du nœde i vers le nœde j

L'heuristique repose sur l'interface d'équilibrage de charge de Charm++ qui extrait la charge de chaque cœur ($L(i)$) et le graphe de communication des tâches. La charge du cœur nous permet d'avoir un aperçu de la façon dont les processeurs sont utilisés par l'application, alors que le graphe de communication donne un aperçu du placement des données sur la machine NUMA. Afin de représenter la hiérarchie machine NUMA et sa topologie, nous utilisons le facteur NUMA (obtenu avec *numarch*) qui fournit une bonne estimation de la latence pour accéder les différents nœuds ($NF(i,j)$).

L'alpha dans l'équation est utilisé pour équilibrer les différentes métriques. Pour chaque tâche, l'heuristique évalue la charge d'un cœur et la communication entre le cœur et tâche considérés. Si un cœur est trop lourd, les tâches ne seront pas migrées vers ce cœur si sa communication avec le cœur est trop importante ($-M(k,i)$). En outre, nous considérons aussi la communication de la tâche avec tous les autres cœurs ($M(k,j)$). Cela est important car quand une tâche migre vers un cœur différent, ses coûts de communication avec tous les autres tâches sont affectés ($\sum_{j!=i}^{nproc}$(M(k,j) × NF(i,j))). Par conséquent, elle peut avoir un impact important dans la performance globale de l'application. Enfin, plus W est petit, plus grane est la possibilité de k à migrer vers le cœur i.

B.8 Résultats

B.8.1 Machines multi-cœur

Afin de faire nos expériences et d'évaluer la performance de *Minas*, nous avons sélectionné trois machine multi-cœur avec des caractéristiques NUMA. Dans cette section, nous présentons ces trois machines et décrivons leurs principales caractéristiques et leurs différences architecturales.

- **AMD8x2**: huit dual-cœur AMD Opteron 875 processeurs. Les cœurs ont cache privé L1 (64 KB) et L2 (1 MB) et aucune cache partagé.
- **Intel4x24**: seize six-cœur Intel Xeon X7460 processeurs. Chaque cœre a une cache privée L1 (32 KB). Chaque deux cœres partagent le cache L2 (256 KB).

Tous les cœurs de un processeur partagent une cache L3 cache (24 MB).
– **Intel4x8**: quatre eight-cœur Intel Xeon X7560 processeurs. Chaque cœre a deux niveaux de cache privé, L1 (32 KB) et L2 (256 KB). Tous les c œurs de un processeur partagent une cache L3 cache (24 MB).

Table B.1: NUMA multi-cœur machines.

Caractéristique	**AMD8x2**	**Intel4x24**	**Intel4x8**
Nombre de cœres	16	96	32
Nombre de processeurs	8	16	4
NUMA nodes	8	4	4
Clock (GHz)	2.22	2.66	2.27
Dernier niveau de cache (MB)	1 (L2)	16 (L3)	24 (L3)
DRAM (GB)	32	192	64
Bande Passante (GB/s)	9.77	4.1	35.54
facteur NUMA (Min;Max)	[1.1; 1.5]	[2.2; 2.6]	[1.36; 3.6]

Tous les machines fonctionnent sous le systéme d'exploitation Linux (kernel 2.6.32) avec GNU Compiler Collection et Intel C Compiler.

Le tableau B.1 résume les caractéristiques matérielles de ces machines. La bande passante mémoire (obtenu avec du Stream - opération Triad [Mccalpin 1995]) et Facteur NUMA sont également présentés dans ce tableau. Les facteurs NUMA sont indiqués en utilisant des intervalles, ce qui signifie le coût minimum et maximale aux accès à la mémoire.

B.8.2 Évaluation des applications OpenMP

Dans cette section, nous présentons l'évaluation de performance de *Minas* sur deux applications réelles de la géophysique. Ces applications permettent aux scientifiques de mieux comprendre les caractéristiques d'une région géographique. Tout d'abord, nous introduisons l'application Ondes 3D qui effectue la simulation de la propagation des ondes sismiques [Dupros 2008, Dupros 2009, Ribeiro 2010c]. Après cela, nous présentons ICTM qui permet de classifier une région géographique en considérant ses caractéristiques [Castro 2009b, Ribeiro 2009a]. Ondes 3D et ICTM sont des applications représentatives des besoins avec une consommation importante de mémoire qui demande une latence faible et une forte bande passante mémoire pour les accès mémoire.

Ondes 3D est une application parallèle qui simule la propagation des ondes sismiques sur une région basé sur la discrétisation en différences finies [Dupros 2008, Dupros 2009]. Elle a été développée par BRGM (www.brgm.fr) et est principalement utilisée pour l'analyse des mouvements forts et de l'évaluation des risques sismiques. La particularité de cette application est de considérer un domaine de calcul fini même si le domaine physique est sans bornes. Par conséquent, l'utilisateur doit définir les conditions aux limites numériques spéciales afin d'absorber l'énergie sortante.

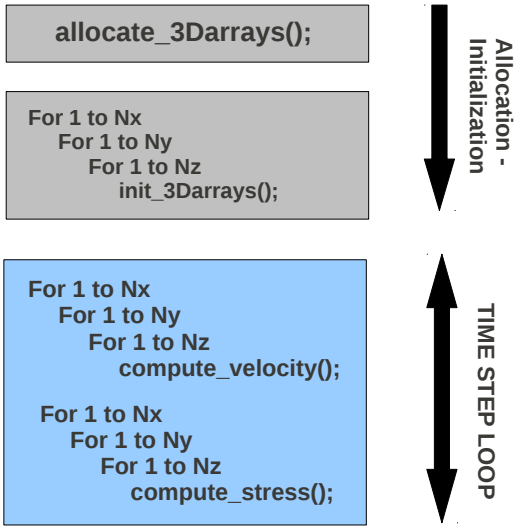


Figure B.5: Application Ondes 3D.

Ondes 3D a trois étapes principales: allocation des données, initialisation des données et calcul de propagation (composé de deux boucles de calcul). Pendant les deux premières étapes, les tableaux qui représente une simulation sont alloués dynamiquement et initialisés. Ces deux étapes sont très importantes parce que les données sont physiquement touchées et placées dans les bancs mémoire de la machine. Au cours de la dernière étape, les deux boucles calculent la vitesse et le stress de la propagation des ondes sismiques. Dans toutes les étapes, les tableaux sont accessibles en mode écriture seule, en lecture seule et en lecture/écriture. Une autre caractéristique importante d'Ondes 3D, est l'accès mémoire régulier aux données. Par un accès régulier, nous voulons dire que les tâches accèdent toujours aux mêmes éléments des tableaux dans le même ordre. La figure B.5 présente un schéma de la demande avec ses trois étapes. Ondes 3D a seulement accès à la mémoire courte distance, seulement quelques éléments du tableau sont nécessaires pour le calcul.

Nous avons réalisé des expériences avec une taille de problème de 2,6 Go (ne tenant pas dans les mémoires cache) et nous utilisons une tâche par cœur. De plus, nous compilons le code de l'application avec la version GCC 4.4.4. Nous comparons le mécanisme *MAi* avec des solutions pour l'affinité mémoire de Linux, le *first-touch*, le *numactl* et le *libnuma*. La version originale d'Ondes 3D s'appuie sur l'allocation dynamique de mémoire, nous ne considérons pas *MApp* sur ces expériences. Des expériences avec différentes tailles du problème et d'autres machines NUMA sont présentées dans [Dupros 2009, Ribeiro 2010b].

En ce qui concerne les solutions Linux pour l'affinité mémoire, nous avons modifié le code source de l'application et/ou ses paramètres exécutions. Pour utiliser *first-touch* et *libnuma*, nous avons changé la répartition des données et l'initialisation. Dans le cas de *first-touch*, nous avons deux versions, l'initialisation nommé maître

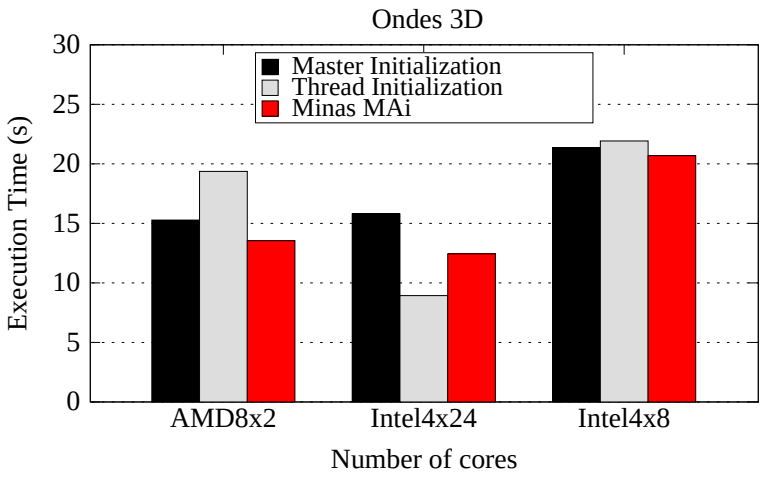


Figure B.6: Temps d'execution (s) pour Ondes 3D.

et un autre nommé initialisation par les tâches. Dans l'initialisation maître (version originale du code), seule la tâche maître initialise tous les tableaux alors que dans l'initialisation de la tâche, chaque tâche initialise ses propres données (notre modification). Pour la version avec *libnuma*, nous avons alloué des données avec les fonctions *numa_ alloc()* et *numa_ alloc_ interleaved()*. Pour le *numactl*, nous utilisons la version initialisation par les tâches de l'application avec l'option *physcpubind* pour d'éviter tout ordonnancement des tâches par Linux. De cette façon, nous garantissons une affinité mémoire parce que les tâches vont toucher leurs propres données et restent dans le nœud où les données qu'ils ont touchée en premier. Nous avons également utilisé l'option *interleaved* de *numactl*, afin de fournir une bonne bande passante pour les tâches.

ICTM est un modèle multi-couche pour la catégorisation des régions géographiques qui utilise plusieurs caractéristiques de la région (relief, végétation, climat, etc.) Le modèle a d'abord été proposé dans [de Aguiar 2004] et la version OpenMP du modèle a été proposée en [Castro 2009a].

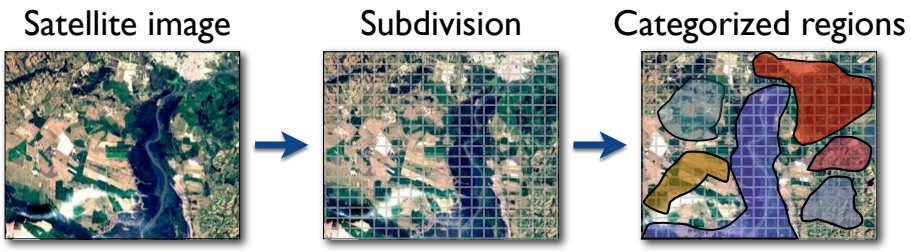


Figure B.7: Entrée and sortie de l'application ICTM.

Le nombre de caractéristiques qui devrait être étudié par ICTM détermine le

nombre de couches du modèle. Dans chaque couche, une analyse différente de la région est effectuée. Les données d'entrée sont extraites à partir d'images satellite (Figure B.7), dans lequel l'information est donnée sur certains points de référence par leur latitude et longitude. Les données extraites sont représentées par une matrice à deux dimensions de la superficie totale en petit sous-zones rectangulaires. Afin de classer les régions de chaque couche, ICTM exécute cinq phases différentes. Chaque phase accède aux matrices spécifiques qui ont déjà été calculées et génère une nouvelle matrice à deux dimensions comme résultat du calcul. Selon la phase, les accès à d'autres matrices peuvent être réguliers ou irréguliers [Castro 2009b]. Comme le montre la figure B.8 (a), l'algorithme utilise essentiellement des boucles imbriquées avec des accès courtes et longues distance sur les matrices (Figure B.8 (b)) lors des phases de calcul. Les accès mémoire à courte distance sont effectuées sur les voisins immédiats (un élément), alors que les accès mémoire à longue distance sont effectuées en utilisant N voisins, où N est définie par le modèle.

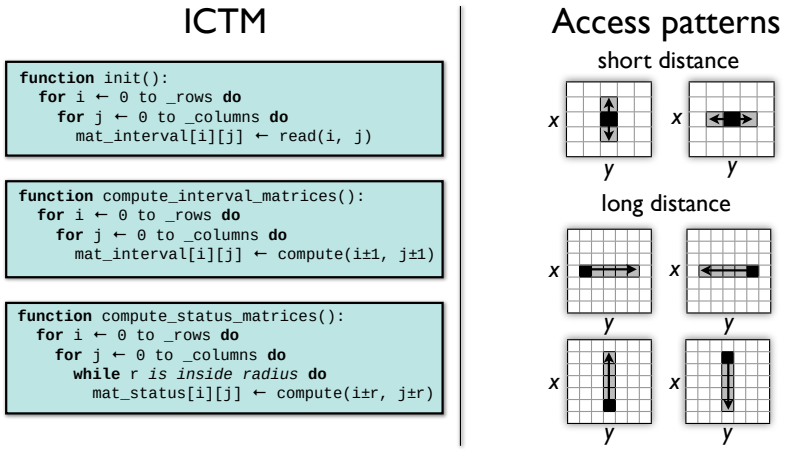


Figure B.8: Application ICTM.

Nous avons effectué des expériences avec un problème de 2 Go et nous utilisons un tâche par cœur de la machine. De plus, nous compilons le code de l'application avec GCC. Nous comparons les mécanismes de *Minas* (*MAi* et *MApp*) avec la solution standard pour l'affinité mémoire sur Linux, le *first-touch*. Des expériences avec différentes tailles du problème, d'autres machines NUMA et des comparaisons avec d'autres mécanismes d'affinité mémoire sont présentées dans [Castro 2009b, Ribeiro 2009b, Ribeiro 2010b].

Pour utiliser la solution *first-touch* et *Minas MAi* sur ICTM, nous avons modifié le code source de l'application. Dans le cas du *first-touch*, nous avons inclus une initialisation parallèle de toutes les matrices utilisées par ICTM. Considérant *MAi*, nous avons ajouté des fonctions de l'interface pour la gestion des données telles que *mai_ alloc()* et politiques mémoire. Les résultats de *MAi* ont été obtenus en appliquant la politique de la mémoire la plus adaptée pour chaque tableau. En

fonction de la phase d'application et la plate-forme, nous avons choisi l'une des politiques de la mémoire, *cyclic neighbors, prime mapp, skew mapp* et *bind block*. Les trois premières politiques sont idéales pour les données partagées en mode lecture sur les plates-formes NUMA qui ont un faible facteur NUMA, car ils permettent plus de débit pour l'accès aux données. Nous avons utilisé ces politiques pour les matrices utilisées dans l'étape de l'intervalle. La politique mémoire *bind block* est adaptée pour les phases régulières où les tâches accèdent toujours au même ensemble de données. Politiques *bind block* sont également indiquées pour les plates-formes avec un haut facteur NUMA, puisque dans ce cas il est important d'éviter les accès distants. La figure B.9 (a) présente l'extrait d'ICTM avec des fonctions *MAi*. Dans cet extrait, toutes les modifications ont été effectuées manuellement.

MAi

```
mai_init();

mat_interval = mai_alloc_2D(_rows,_cols,
                  sizeof(float),FLOAT);
mat_status = mai_alloc_2D(_rows, _cols,
                  sizeof(float),FLOAT);

mai_bind_rows(mat_interval);
mai_bind_rows(mat_status);

function_init();

function_compute_interval_matrices();

//change memory policy for mat_status
mai_skew_mapp(mat_status);
function_compute_status_matrices();

mai_final();
```

(a)

MApp

```
mai_init();

mat_interval = mai_alloc_2D(_rows,_cols,
                  sizeof(float),FLOAT);
mat_status = mai_alloc_2D(_rows, _cols,
                  sizeof(float),FLOAT);

mai_cyclic(mat_interval);
mai_cyclic(mat_status);

function_init();

function_compute_interval_matrices();

function_compute_status_matrices();

mai_final();
```

(b)

Figure B.9: ICTM avec *Minas*: (a) version *MAi* (b) version *MApp*.

Pour la solution MApp, le code a été pré-traité avec CUIA et modifié par MApp pour appliquer les politiques mémoire sur le code source ICTM. MApp insère dans le code source des fonctions d'allocation et politiques mémoire pour tous les tableaux globale partagée d'ICTM. Il ne considère pas les variables privées et les temporaires créés au sein des fonctions d'ICTM. Pour les variables globales et compte tenu des plates-formes, l'heuristique de *MApp* a sélectionné les politiques mémoire tels que *bind block* et *cyclic*. La figure B.9 (b) présente l'extrait d'ICTM, avec les adaptations *MApp*. Nous pouvons observer que différemment de la version *MAi*, celui-ci ne applique des politiques qu'au début de l'application. Aucune modification n'est effectuée entre les phases de l'application.

La figure B.10 représente les accélérations pour ICTM sur les plates-formes AMD8x2 et Intel4x8. On peut remarquer que les mécanismes de *Minas* ont surclassé

le *first-touch* solution sur la plate-forme AMD8x2. Pour la plate-forme Intel4x8, nous pouvons observer que *MAi* a obtenu les meilleurs résultats alors que *MApp* a échoué pour un grand nombre de cœurs, toutefois, sur les deux plates-formes, *MApp* a présenté de meilleurs résultats que la solution *first-touch*.

ICTM a cinq étapes différentes avec différents accès mémoire, il est donc difficile de profiter de *first-touch* pour placer les données sur la machine. *first-touch* distribue des données sur la machine en regardant le premier accès sur les données par des tâches. Pour cette raison, un accès mémoire différent sur n'importe quel étape de calcul de l'application peut générer des accès les plus coûteux ou des problèmes d'équilibrage de charge. En outre, le ICTM a une étape qui dépend de la bande passante et *first-touch* ne considère que la latence pour placer les données sur les bancs mémoire de la machine. Après une analyse approfondie des résultats pour chaque étape (présenté dans [Castro 2009a, Castro 2009b]), nous avons observé que *first-touch* obtient de meilleurs résultats sur les phases qui ont accès à la mémoire régulière de manière similaire à la phase d'initialisation.

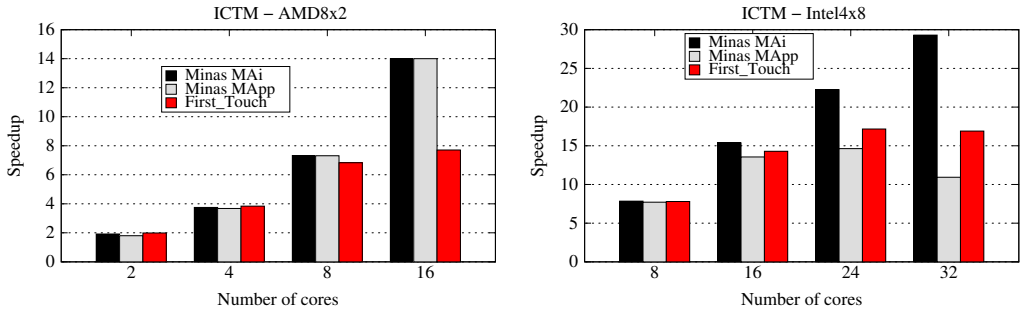


Figure B.10: Performances d'ICTM sur AMD8x2 et Intel4x8.

Dans le graphiqueB.10 (a), nous observons que *MAi* et *MApp* ont obtenu des performances similaires pour ICTM sur la machine AMD8x2. Dans ce cas, la machine a un petit facteur NUMA (les accès à distance ne sont pas chers) et les politiques *cyclic* (utilisé par *MApp*) fournit une bande passante beaucoup plus haut pour les tâches. On peut aussi remarquer que dans cette plate-forme, le *first-touch* ne passe pas à l'échelle lorsqu'on augmente le nombre de tâches. Cette plate-forme a des problèmes de bande passante et avec un grand nombre de cœurs, il est important de fournir aux tâches des bonnes performances pour accéder aux données. La figure B.10 (b) montre que le mécanisme *MAi* a obtenu des bonnes performances et d'évolutivité pour tous les nombre de cœurs. *MApp* n'a pas obtenu de bons résultats sur le Intel4x8 parce que son heurietique considère que les politiques standard *cyclic* et *bind block*. Il ne fait pas usage d'autres politiques mémoire cycliques comme *skew mapp, cyclic neighbors* (utilisé dans le *Minas MAi* solution).

B.8.3 Évaluation sur les applications Charm++

Dans cette section, nous présentons l'évaluation des performances des deux mécanismes d'affinité mémoire développés pour le système parallèle Charm++ en utilisant des composants *Minas*. Dans notre évaluation de performance, nous utilisons trois benchmarks de Charm++, le kneighbor, le 2D moléculaire et le jacobi 2D [PPL-Charm++ 2011]. Le kneighbor et 2D moléculaire nous permettre d'évaluer les placements des données de *MAi* (+ maffinity) pour des applications avec des caractéristiques différentes (la consommation mémoire et la communication) et des différentes exigences (la bande passante mémoire et la latence). Le jacobi est utilisé pour évaluer la performance du équilibreur de charge NUMA proposé dans cette thèse. Il présente des caractéristiques de déséquilibre de charge. En outre, Jacobi a un rapport important de communication et de calcul. Pour cette évaluation, nous utilisons les trois plates-formes NUMA décrites dans le chapitre B.8.1 et charm++ 6.2.0 avec une installation multi-cœur.

Pour les résultats suivants, nous avons utilisé une tâche par cœur sur toutes les exécutions. Nous comparons les résultats obtenus avec le mécanisme +**maffinity** à ceux obtenus par l'utilisation du mécanisme +**setcpuaffinity** [Mei 2010].

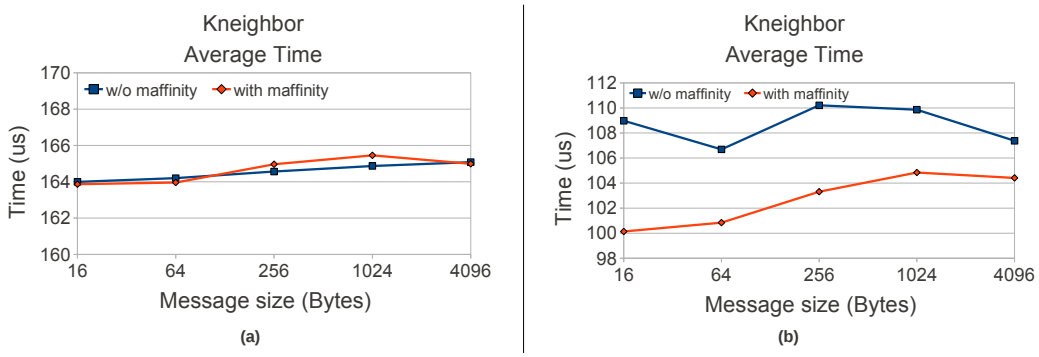


Figure B.11: Temps d'execution (us) Kneighbor: (a) AMD8x2 (b) Intel4x8.

Dans la figure B.11, nous pouvons observer que pour le kneighbor que l'exécution avec affinité mémoire a obtenu de meilleurs résultats en moyenne pour les deux machines avec des tailles de message différent. Toutefois, les gains sont plus expressifs dans la machine Intel4x8 NUMA (jusqu'à 8%). Dans cette machine le facteur NUMA est plus élevé et, par conséquent, l'impact de l'amélioration de l'affinité mémoire est plus important. Pour ce critère et compte tenu des machine sélectionnées, il est important de réduire la latence pour obtenir des données. Pour ces expériences, nous avons utilisé le support d'affinité de CPU de Charm++ et puis, nous avons également placé des données sur le nœud où les tâches sont en cours d'exécution. Par conséquent, la performance a été meilleure, car nous avons amélioré la localité des données.

La figure B.12 indique la durée moyenne pour kneighbor lorsqu'il est exécuté sur la plate-forme Intel4x24. Pour ces expériences, nous avons utilisé 24 et 64 cœurs

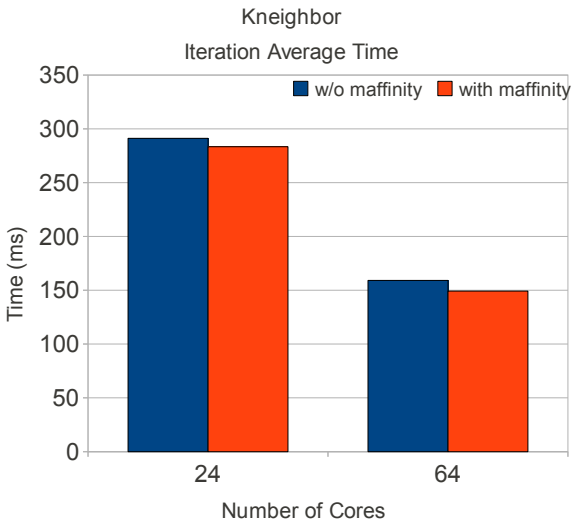


Figure B.12: Temps d'execution Kneighbor sur Intel4x24.

et les tailles des messages de 1024 octets. Nous pouvons observer que les résultats avec l'affinité mémoire sont similaire à sans affinité pour les 24 cœurs. Dans ce cas, l'application s'exécute dans un seul nœud NUMA et, par conséquent, aucune amélioration peut être générée par l'utilisation de +**maffinity**. Toutefois, pour les 64 cœurs nous constatons un amélioration jusqu'à 6%. +**maffinity** définit les nœuds et la politique qui doivent être utilisés pour l'exécution à fin de réduire les pénalités NUMA.

Dans le tableau B.2, nous présentons les temps d'exécution d'une étape (ms) de moléculaire 2D lorsqu'il est exécuté avec +**setcupaffinity** et +**maffinity**. En général, +**maffinity** a présenté quelques légères améliorations par rapport à +**setcupaffinity**, jusqu'à 5% de gains. Le meilleur résultat a été obtenu sur la machine NUMA Intel4x8 lorsque quatre nœuds de la machine ont été utilisés. La différence de temps d'itération est principalement liée à la réduction des accès à distance effectués simultanément par les tâches sur le même banc mémoire. L'utilisation de la politique *cyclic neighbors* nous a permis d'assurer un meilleur équilibrage de la charge des pages mémoire sur les bancs mémoire. En utilisant cette politique des pages mémoire sont plus disponibles au même temps pour des accès simultanés.

Table B.2: Temps d'execution (ms) - une iteration Molecular 2D

	AMD8x2		**Intel4x8**	
	8 Cœurs	16 Cœurs	16 Cœurs	32 Cœurs
w/o maffinity	131.46	68.84	1083.74	698.67
maffinity	125.08	67.43	1038.80	692.06

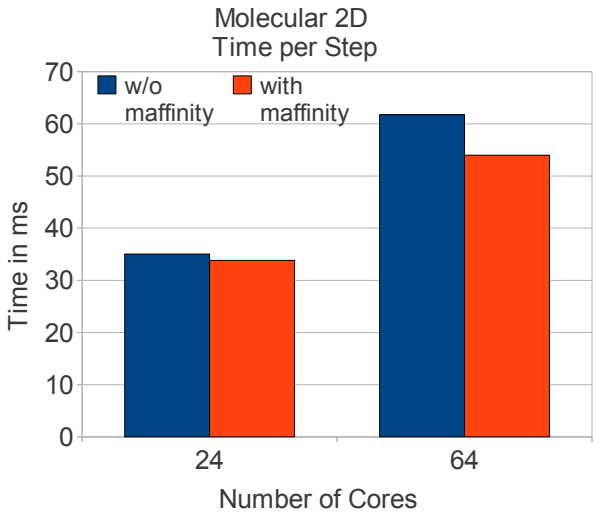


Figure B.13: Temps moyen d'itération pour Molecular 2D sur Intel4x24.

La figure B.13 montre le temps par étape pour le Moléculaire 2D sur la plateforme Intel4x24. Dans ces expériences, nous avons utilisé 24 et 64 cœurs. Les résultats obtenus avec +**maffinity** pour les deux nombre de cœurs ont surmonté les performances obtenues sans le support d'affinité de mémoire (jusqu'à 12% de gains). Compte tenu des caractéristiques de l'application, nous avons appliqué le politique *cyclic neighbors* qui distribue les données sur les bancs mémoire voisins. Cette politique s'adapte bien à la répartition du travail par défaut utilisé pour cette application qui est une stratégie de type round-robin. Plus de résultats avec le +**maffinity** sont présentés dans [Ribeiro 2010d].

Nous présentons maintenant l'évaluation de l'équilibreur de charge pour les machines NUMA proposé dans cette thèse. Nous comparons les résultats obtenus avec **NumaLB** à ceux obtenus sans équilibrage de charge et avec deux autres l'équilibreur de charge, le Metis [Karypis 1995] et Greedy [PPL-Charm++ 2011].

Similaires à NumaLB, l'équilibreur de charge Metis considère également les coûts de communication pour équilibrer la charge entre les noyaux de la machine. Il utilise des mécanismes de partition de graphique de la bibliothèque Metis pour créer un schéma qui représente la communication. Contrairement à cette stratégie, l'équilibreur de charge Greedy ne tient pas compte des caractéristiques de la communication pour effectuer l'équilibrage de charge. Sa stratégie est d'envoyer les charges les plus lourdes sur le processeur le moins chargé, jusqu'à ce que l'équilibrage de charge soit atteint. Nous avons d'abord décrit la performance globale obtenue avec NumaLB et après nous l'avons comparée avec les résultats obtenus pour des autres équilibreurs de charge. Après cela, nous présentons des statistiques de NumaLB qui nous permettent d'évaluer les coûts généraux pour extraire la topologie de la machine.

La figure B.14 montre les accélérations pour le Jacobi 2D avec et sans l'équilibreur

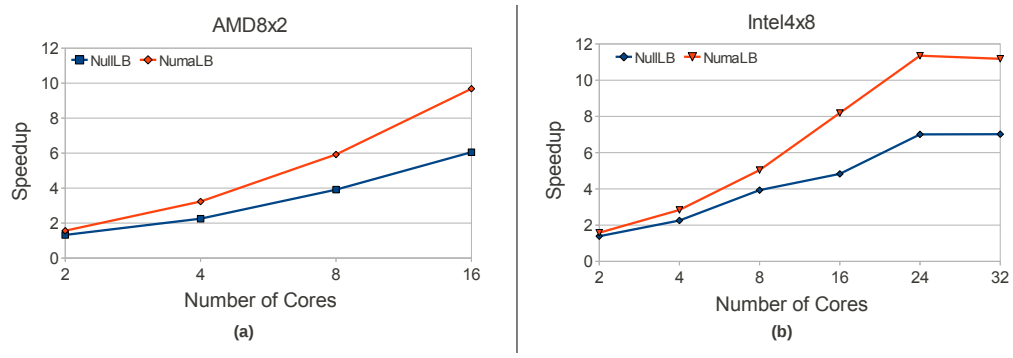


Figure B.14: Jacobi 2D Speedups: (a) AMD8x2 (c) Intel4x8.

de charge NumaLB. Nos résultats ont montré que l'équilibreur de charge NumaLB obtient des améliorations de performances allant jusqu'à 68%, avec une moyenne de 24%, sur une version sans équilibreur de charge pour Jacobi 2D sur des machines NUMA. Nous pouvons observer que sur les deux machines, Jacobi a présenté une meilleure évolutivité avec l'équilibreur de charge NumaLB. Dans le cas NumaLB, les migrations des objets en considérant le facteur NUMA évite toute communication à longue distance entre les tâches. Pour cette raison, l'impact NUMA sur l'exécution d'application est réduit.

Nous avons observé le même comportement sur les deux machines, mais dans le Intel4x8 l'amélioration des gains est plus élevée parce qu'il présente un haut facteur NUMA. Comme les accélérations ont été calculées en considérant les temps d'exécution de Jacobi 2D, ils nous permettent de confirmer que les coûts généraux pour extraire la topologie machine ne dégradent pas la performance globale de l'application. En outre, ces résultats nous permettent également de confirmer que la stratégie NumaLB a un temps de réponse réduit. Par conséquent, l'équilibrage de charge est effectué sans un impact important sur la durée d'exécution globale des applications.

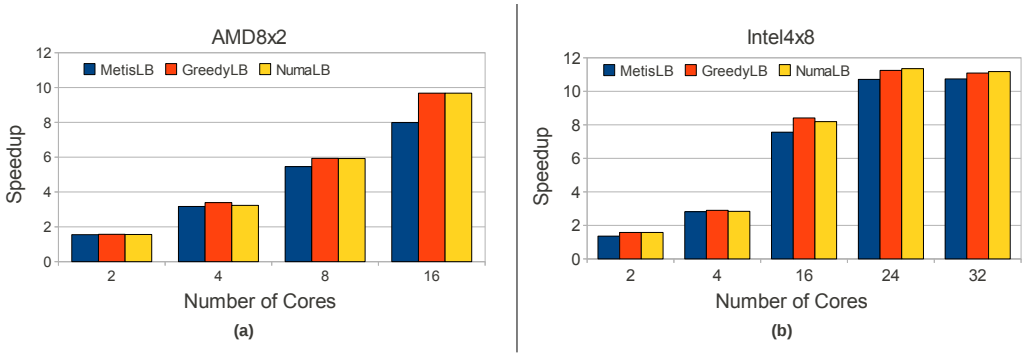


Figure B.15: Jacobi 2D Speedups: (a) AMD8x2 (c) Intel4x8.

Afin de vérifier l'efficacité de NumaLB, nous avons également comparé ses performances avec MetisLB et GreedyLB sur le AMD8x2 et machines Intel4x8.

La figureB.15 présente l'accélération obtenue avec NumaLB, MetisLB et GreedyLB pour Jacobi 2D. Dans l'ensemble, la stratégie NumaLB a joué jusqu'à 16% de mieux que l'équilibreur de MetisLB. Malgré MetisLB considère les caractéristiques de communication de l'application pour effectuer l'équilibrage de charge, il ne prend pas en compte la topologie de la machine sur sa stratégie. MetisLB ne peut pas prendre en compte les coûts de la latence d'accès d'une plate-forme NUMA. Opposé à la comparaison avec MetisLB, par rapport à l'équilibreur de charge GreedyLB, NumaLB a présenté des performances similaires. Dans le cas de la machine AMD8x2, cela est dû à un facteur NUMA petit. Dans cette machine, le temps d'accès pour les demandes de données à distance sont faibles quand comparé avec des demandes de données locales. La machine Intel4x8 dispose d'un grand cache L3 partagé qui favorise l'équilibrage de charge de GreedyLB. Les grandes caches L3 partagé sur la machine Intel4x8 réduit l'accès à la mémoire locaux et distants. Par contre, NumaLB n'a pas diminué les performances de Jacobi 2D et pour un grand nombre de cœurs il a présenté de meilleurs résultats.

Table B.3: Statistiques d'exécution.

	AMD8x2		**Intel4x8**	
	Temps d'Init. (s)	Strategie (ms)	Temps d'Init. (s)	Strategie (ms)
MetisLB	0.22	0.502	0.106	0.549
GreedyLB	0.17	0.055	0.101	0.063
NumaLB	0.19	0.301	0.100	0.264

La principale contribution de cette thèse dans la conception de l'équilibreur de charge NumaLB est la représentation de la hiérarchie et la topologie NUMA de la machine fournie par le module *numarch* de *Minas*. Par conséquent, il est également important d'évaluer les surcoûts imposés par *numarch* pour extraire les informations de la machine. Pour ce faire, nous avons choisi deux indicateurs de performance, le temps d'initialisation pour charger les informations de la machine et le temps pris par NumaLB pour exécuter sa stratégie.

Tableau B.3 rapporte chacun de ces paramètres pour MetisLB, GreedyLB et NumaLB. Nous pouvons observer que le temps passé par NumaLB pour initialiser l'équilibreur de charge est réduit de 13% par rapport au temps dont le MetisLB et augmenté de 13% par rapport au temps de la GreedyLB. Considérant le temps nécessaire pour calculer la stratégie NumaLB, il a donc été meilleur que MetisLB mais moins performant que GreedyLB. Comme mentionné avant, GreedyLB ne considère pas les informations de communication de l'application pour effectuer l'équilibrage de charge: il ne considère que la charge de chaque processeur. Par conséquent, sa stratégie est moins coûteuse à calculer que MetisLB et NumaLB. Toutefois, cette

différence ne réduit pas les performances globales de NumaLB. Une évaluation complète de la performance NumaLB avec d'autres repères et les comparaisons sont présentées [Pilla 2011a].

B.9 Conclusion

Dans ce dernier chapitre, nous concluons la thèse présentant ses principaux objectifs, les contributions sur la gestion d'affinité mémoire pour machines NUMA et les perspectives générer par ce travail.

B.9.1 Objectifs de la thèse

Au cours des dernières années, les puces multi-cœur sont devenues une tendance dans la conception de processeurs de machines à mémoire partagée. Cependant, comme le nombre de cœurs par puce augmente, l'accès à la mémoire partagée devient un goulot d'étranglement. Les cœurs accèdent à la mémoire partagée, ce qui a pour effet de surcharger la mémoire et de rendre machines parallèles moins performantes. Dans ce contexte, pour atténuer le problème de mémoire, l'architecture NUMA a été employé dans les machines multi-cœur.

Ces machines NUMA multi-cœur sont généralement construites avec des contrôleurs mémoire sur le puce, qui fournit l'abstraction d'une mémoire partagée unique pour les cœurs de la machine. Bien que cette conception réduise les conflits de mémoire pour les machines à mémoire partagée, il peut potentiellement augmenter la latence d'accès mémoire et dégrader la bande passante. Cela est dû au fait que sur les machines NUMA, la mémoire partagée est physiquement répartie en plusieurs blocs de mémoire qui sont inter-connectés par un réseau. L'utilisation de l'affinité mémoire devient donc essentielle pour assurer de bonnes performances dans telles machines. Dans les machines NUMA, les mécanismes d'affinité mémoire, essayent de conserver les données à proximité des cœurs afin de réduire les coûts d'accès mémoire pour les applications parallèles.

Considérant des machines à mémoire partagée, il existe plusieurs langages de programmation qui peuvent être utilisés pour développer des applications parallèles. Par exemple, l'API OpenMP et Charm++ sont des exemples de support à la programmation pour les développeurs. Cependant, la plupart des langages parallèles ne tiennent pas compte des machines multi-cœur avec l'architecture NUMA. Ces langages ne disposent pas d'un support pour contrôler l'affinité mémoire, qui finit par être gérée par le système d'exploitation ou même par le programmeur. Les travaux développés dans cette thèse sont liés à la gestion d'affinité mémoire pour les multi-cœurs NUMA pour des langages parallèles. L'objectif principal est de cacher au programmeur la complexité du contrôle de l'affinité mémoire dans un code source des applications parallèles. Les caractéristiques plate-forme NUMA et les accès à mémoire d'application sont utilisés afin de fournir une gestion de la mémoire à grain fin.

B.9.2 Contributions

Afin de renforcer l'affinité mémoire sur des machines multi-cœur NUMA, nous avons proposé dans cette thèse l'environnement *Minas*. Il permet de contrôler l'affinité mémoire par le placement de données et des tâches sur les plates-formes NUMA. Les mécanismes de placement des données et de tâche prennent en compte les caractéristiques la machine et les caractéristiques de l'application. La topologie de la machine et de la représentation hiérarchie utilisée par *Minas* lui permet d'aborder l'architecture hiérarchique de ces machines. Le préprocesseur *MApp* aide à l'extraction des caractéristiques de l'application qui doivent être considérées avant le placement des données et des tâches. Ces informations sont couplées dans un certain nombre de politiques mémoire et de placement des tâches à l'intérieur de *Minas* pour améliorer l'affinité mémoire.

Minas met en œuvre deux types de politiques de mémoire, le *bind* qui réduit la latence d'accès perçue par les tâches pour obtenir des données et le *cyclic*, qui réduit les conflits de mémoire, fournissant plus de bande passante pour les tâches dans l'obtention des données. Ces politiques de mémoire peuvent être appliquées à différents niveaux des données d'application telles que le tas, la pile et variables. Considérant les mécanismes de placement des tâches à l'intérieur de *Minas*, l'environnement met en œuvre les mécanismes statiques et dynamiques. Le mécanisme statique place les tâches sur les cœurs de la machine avec l'objectif de maximiser le partage cache. Ceci est réalisé en utilisant la topologie de la machine et des traces de mémoire de l'application. Le placement dynamique des tâches est implémenté comme un équilibreur de charge qui utilise la topologie de la machine NUMA et les caractéristiques de la communication de l'application lors de l'exécution.

En considérant *Minas*, nous avons montré que la gestion d'affinité mémoire peut être faite de manière explicite ou de façon automatique. Dans ce contexte, les développeurs qui connaissent leurs applications peuvent contrôler manuellement l'affinité en fournissant quelques indications sur les modèles d'accès à la mémoire pour *Minas*. Dans ce cas, le programmeur inclut des fonctions de l'interface *MAi* dans le code source de l'application pour allouer et de placer des données. En utilisant ces fonctions, *Minas* place des données d'application sur les nœuds NUMA de la machine. Les développeurs qui n'ont pas une connaissance précise, a priori des accès mémoire pour une application, peuvent automatiquement contrôler l'affinité à l'aide de *MApp*. Ce mécanisme transforme le code source de l'application en utilisant les caractéristiques de la machine et de l'application extraites au moment de la compilation. Ainsi, nous avons montré que ces mécanismes peuvent être combinés afin de renforcer encore plus l'affinité mémoire et, par conséquent, la performance des applications parallèles.

Nous avons utilisé les composants de *Minas* dans quatre interfaces parallèle/langages pour développer des applications parallèles. L'exploration des composants *Minas* dans chaque interface parallèle/langage a été faite à différents niveaux. Considérant le niveau de langage, nous avons montré que *Minas* pouvait être utilisé lors de la compilation pour contrôler l'affinité mémoire pour les applications OpenMP.

Minas supporte les principaux compilateurs (Intel, GNU, PGI,...). Nous avons également exploré *Minas* pour contrôler l'affinité mémoire dans des environnements dynamiques tels que le système d'exécution Charm++ et l'environnement AMPI. Dans le cas de Charm++ et AMPI, une intégration des composants *Minas* a été appliquée et offre aux utilisateurs un support d'affinité mémoire transparent. Enfin, au niveau algorithmique, nous avons intégré *Minas* pour fournir un support d'affinité mémoire pour un environnement à base de squelettes *OpenSkel*. En utilisant les informations fournies par les squelettes, *Minas* est capable de traiter des données et de placer des tâches pour une application basée sur des squelettes.

B.9.3 Perspectives

Les approches pour l'affinité mémoire proposées dans cette thèse conduisent à un certain nombre de perspectives.

Améliorer le modèle de la machine NUMA: Le modèle actuel que nous avons défini pour représenter la topologie et la hiérarchie mémoire NUMA peut être étendu avec plus de détails sur l'architecture mémoire. Par exemple, la représentation de la hiérarchie mémoire s'appuie sur le comportement des accès mémoire en lecture. Ici, la prise en compte des opérations d'écriture permettrait d'affiner encore le modèle. Dans ce cas, nos représentations de la hiérarchie mémoire seraient capables de modéliser mieux les différents accès à la mémoire d'une application. En considérant la bande passante, notre modèle pourrait donc s'appliquer aux cas d'applications avec plusieurs tâches qui accèdent de manière concurrente à la mémoire générant ainsi de la contention. Ces nouvelles informations permettraient à *Minas* d'améliorer le placement des données en tenant compte de la saturation du réseau d'interconnexion. Une autre évolution possible est de modéliser les différents niveaux de mémoire cache pour fournir à *Minas* les surcoût liés à la communication intra-nœud. L'utilisation de compteurs de performance des processeurs pour récupérer les informations des applications lors de l'exécution est également une perspective à ce travail de thèse. La topologie de la machine pourrait être renforcée pour soutenir les systèmes d'exploitation autres que Linux. Néanmoins, notre modèle pourrait être intégré dans les outils de niveau de l'utilisateur tels que la bibliothèque hwloc.

Obtenir plus d'information de l'application: Dans *Minas*, les modifications automatiques pour les applications parallèles pourraient être étendues avec un support pour plusieurs langages. Dans cette thèse, nos proposition pour l'approche automatique n'ont été appliquées que sur des applications écrites en C avec OpenMP. Dans le cas d'OpenMP, nous envisageons d'étendre notre préprocesseur pour supporter des applications écrites en Fortran. En effet, de nombreuses applications scientifiques OpenMP sont écrites en Fortran. Par conséquent, un support pour ce langage est important dans le contexte du calcul haute performance. Vu les demandes OpenMP, des informations telles que l'accès mémoire sur les variables à l'intérieur des régions parallèles doivent être considérées. Cette information permettrait *Minas* de mieux gérer la distribution des données sur la machine. Par

exemple, les accès indirects ou les séquentielles offrent à *Minas* les informations de la façon dont les données sont accessibles à l'intérieur d'une région parallèle à la compilation. Dans ce contexte, le support d'autres systèmes de programmation parallèle comme Charm++/ AMPI constitue également une perspective à cette thèse. De plus, *Minas* pourrait aussi profiter des informations d'exécution de l'application pour corriger le placement des données et des tâches.

Support pour les architectures hybrides: Les architectures multi-cœur sont de plus en plus complexes. Cette complexité provient principalement de la hiérarchie mémoire et des hiérarchies de cache. De plus, les plates-formes de calcul sont également équipées de processeurs graphiques (GPU) qui disposent d'une puissance de calcul importante. Les processeurs graphiques (GPU) complexifient l'architecture et rendent délicates la programmation. Les éléments de la machine se partagent un espace global de mémoire, mais encore une fois avec des coûts différents pour y accéder. Les processeurs graphiques (GPU) sont aujourd'hui connectés par un bus d'entrée-sortie. Pour cette raison, un support d'affinité mémoire est indispensable pour réduire les coûts de communication entre les GPU et des cœurs. Par conséquent, une perspective à envisager est d'étendre *Minas* pour supporter les plates-formes hybrides de calcul. Dans ce contexte, les mécanismes de base de l'environnement *Minas* tels que l'allocation de mémoire, le placement des données et l'ordonnancement des tâches devraient tenir compte des différents éléments architecturaux pour renforcer l'affinité mémoire pour les applications parallèles. Par conséquent, notre modèle de machine devrait être étendu afin de considérer les machines hybrides. Nos mécanismes pour contrôler la mémoire devraient donc prendre en charge les espaces mémoire différents (mémoire GPU et mémoire de la machine) qui composent la mémoire globale et les différentes représentations des structures de données dans la mémoire GPU et mémoire de la machine. De plus, dans le calcul haute performance, le nombre de cœurs par processeurs va encore augmenter, conduisant à des architectures de multi-cœurs. Par conséquent, différents niveaux d'accès non-uniforme à la hiérarchie mémoire sont attendus. Fournir un support pour ces architectures est une perspective à envisager après cette thèse.