HAL
open science

Minas: Memory Affinity Management Framework

Christiane Pousa Ribeiro, Jean-François Méhaut

To cite this version:

Christiane Pousa Ribeiro, Jean-François Méhaut. Minas: Memory Affinity Management Framework. [Research Report] RR-7051, INRIA. 2009. inria-00421546

HAL Id: inria-00421546

https://inria.hal.science/inria-00421546v1

Submitted on 2 Nov 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

inria-00421546, version 1 - 2 Oct 2009

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Minas: Memory Affinity Management Framework

Christiane Pousa Ribeiro — Jean-François Méhaut

N° 7051

October 2009

Thème NUM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7051--FR+ENG

inria-00421546, version 1 - 2 Oct 2009

inria-00421546, version 1 - 2 Oct 2009

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche GRENOBLE - RHÔNE-ALPES

Minas: Memory Affinity Management Framework

Christiane Pousa Ribeiro, Jean-François Méhaut

Thème NUM — Systèmes numériques
Équipe-Projet MESCAL

Rapport de recherche n° 7051 — October 2009 — 25 pages

Abstract: In this document, we introduce Minas, a memory affinity management framework for cache-coherent NUMA (Non-Uniform Memory Access) platforms, which provides either explicit memory affinity management or automatic one with efficiency and architecture abstraction for numerical scientific applications. The explicit tuning is based on an API called MAi (Memory Affinity interface) which provides simple functions to manage allocation and data placement using an extensive set of memory policies. An automatic tuning mechanism is provided by the preprocessor named MApp (Memory Affinity preprocessor). MApp analyses both the application source code and the target cache-coherent NUMA platform characteristics in order to automatically apply MAi functions at compile time. Minas efficiency and architecture abstraction have been evaluated on two cache-coherent NUMA platforms using three numerical scientific HPC applications. The results have shown significant gains when compared to other solutions available on Linux (First-touch, libnuma and numactl).

Key-words: multi-core architecture, memory affinity, static data, performance evaluation, programming, tuning

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

inria-00421546, version 1 - 2 Oct 2009

Minas: Memory Affinity Management Framework

Résumé : Ce document décrit Minas, un environnement logiciel (framework) pour contrôler l'affinité mémoire sur les architecture NUMA. Ce framework propose une gestion implicite et explicite de l'affinité de mémoire pour les applications scientifiques. Les principales caractéristiques sont l'efficacité et une bonne abstraction de l'architecture. Dans Minas, la gestion explicite de l'affinité mémoire repose sur une interface de programmation (API) appelée MAi (Memory Affinity interface) qui offre des fonctions simples pour gérer l'allocation et le placement des données en utilisant de nombreuses politiques d'affinité. Un mécanisme automatique de contrôle de l'affinité de mémoire est fourni par le préprocesseur Mapp (Memory Affinity preprocessor). Mapp analyse le code source de l'application et connaît les caractéristiques de la plate-forme NUMA afin d'appliquer automatiquement les fonctions du MAi pendant la phase de compilation. L'efficacité de Minas a été évaluée sur deux plates-formes ccNUMA avec trois applications scientifiques hautes performances. Les résultats ont révélé des gains importants par rapport aux autres solutions disponibles sous Linux (First-touch, libnuma and numactl).

Mots-clés : architectures NUMA, affinité mémoire, données statiques, étude de performances, optimisation

inria-00421546, version 1 - 2 Oct 2009

1 Introduction

The increasing number of cores per processor and the efforts to overcome the limitation of classical Symmetric Multiprocessors (SMP) parallel systems remain a problem. Due to this, Non-Uniform Memory Access (NUMA) platforms are becoming very common computing resources for numerical scientific High Performance Computing (HPC). A NUMA platform is a large scale multi-processed system in which the processing elements are served by a shared memory that is physically distributed into several memory banks interconnected by a network. Thus, memory access costs are not symmetric, since the distance between cpus and memory banks may vary (local and remote accesses are possible). The effects of applications memory access costs in such platforms can be reduced through the guarantee of memory affinity [1, 2, 3].

Memory affinity is assured when a compromise between threads and data is achieved by reducing either the number of remote accesses (latency optimization) or the memory contention (bandwidth optimization). In the past, researches have led to many different solutions: libraries, interfaces, tools and memory policies in user or kernel spaces of operating systems. However, most of these solutions demand considerable changes in the application source code considering architecture characteristics. Such solutions are thus not portable, since they do not offer architecture abstraction and consequently, developers must have prior knowledge of the target platform characteristics (e.g., number of processors/cores and number of nodes). Additionally, these solutions were limited to specific NUMA platforms, they did not address different memory accesses (limited set of memory policies) and they did not include optimizations for numerical scientific data (i.e., array data structures).

To overcome these issues, our research have led to the proposal of Minas: an efficient and portable framework for managing memory affinity on cache-coherent NUMA (ccNUMA) platforms. Minas enables explicit and automatic control mechanisms for numerical scientific HPC applications. Beyond the architecture abstraction, this framework also provides several memory policies allowing better memory access control. We have evaluated its portability (architecture abstraction) and efficiency by performing experiments with four numerical scientific HPC applications on two ccNUMA platforms. The results have been compared with three often used solutions on ccNUMAs (*first-touch*, *numactl* and *libnuma* from Linux).

The repport is structured as follows. In Section 2 we describe the previous solutions in memory affinity management for NUMA architectures. Section 3 introduce our framework and its design. We present in Section 4 the evaluation of architecture abstraction of Minas and discuss its performance evaluation. Finally, in the last section we present our conclusions and future work.

2 Memory Affinity Management Solutions

Many different works has been done on memory affinity management for NUMA architectures. Most of them are proposals of new memory policies that have some intelligent mechanism to place/migrate memory pages. Other types of works are proposal of new directives to OpenMP and some support integrated

in the operating system. In this section we present these three groups of related work.

2.1 Memory Affinity Policies

Memory policy approach is the simplest way to deal with memory affinity, since developers do not have to carry about the memory management. In this approach, the operating system is responsible for optimizing all memory allocation and placement [4, 5, 6, 7, 8].

First-touch is the default policy in Linux operating system to manage memory affinity on ccNUMAs. This policy places data on the node that first accesses it [9, 3]. 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* will result in a high number of remote accesses, since threads do not access the same data.

In the work [4], authors present the proposal of a new memory policy named *on-next-touch* for Solaris operating system. This policy allows data migration when threads touch them for the next time. Thus, threads can have their data in the same node, allowing more local access. In this work, the performance evaluation of this policy was done using a real application that has as main characteristic irregular data access patterns. The gain obtained with this solution is about 69% with 22 threads. Currently, there are some proposals of *on-next-touch* memory policy for Linux operating system. For instance, in [5, 6], the authors have designed and implemented the *on-next-touch* policy on such operating system. Its performance evaluation has shown good performance gains only for applications that have a single level of parallelism. When it was applied in nested parallel levels, it was not profitable (threads frequently lost their affinity). Thus, many data migrations were done and this overhead lowered the performance gains.

In [7], the authors present two new memory policies called skew-mapping and prime-mapping. In the first one, allocation of memory pages is performed skipping one node per round. As example, suppose that we have to allocate 16 memory pages in four nodes. The first four pages will be placed on nodes 0,1,2,3, the next four in nodes 1,2,3,0 and so on. The prime-mapping policy works with virtual nodes to allocate data. Thus, after the allocation on the virtual nodes there is a re-allocation of the pages in the real nodes. As scientific applications always work in power of 2, for data distribution, these two policies allows better data distribution. The gains with this solutions are about 35% with some benchmarks.

The work [8] present two algorithms to do page migration and assure memory affinity in NUMA machines. These algorithms use information extracted from kernel scheduler to perform page migrations. The performance evaluation has shown gains of 264% considering existed solution as comparison.

2.2 Memory Affinity with OpenMP Directives

In [10], authors present a strategy to assure memory affinity using OpenMP in NUMA machines. The idea is to use information about schedule of threads and

inria-00421546, version 1 - 2 Oct 2009

data and made some relations between them. The work do not present formal OpenMP extensions but shows some suggestions of how this can be donne and what has to be included. All performance evaluation was donne using tightly-coupled NUMA machines. The results show that their proposal can scale well in the used machines.

In the work [11], authors present new OpenMP directives to allow memory allocation in OpenMP. The new directives allows developers to express how data have to be allocated in the NUMA machine. All memory allocation directives are for arrays and Fortran programming language. The authors present the ideas for the directives and how they can be implemented in a real compiler.

2.3 Memory Affinity with Operating System Support

NUMA support is now present in several operating systems, such as Linux and Solaris. This support can be found in the user level (with administration tools or shell commands and NUMA APIs) and in the kernel level (with system call) [12]. On Linux operating system, a basic support to manage affinity on ccNUMAs and is implemented in three parts: kernel/system calls, a library (libnuma) and a tool (numactl).

The kernel part defines three system calls (*mbind*(), *set_mempolicy*() and *get_mempolicy*()) that allow the programmer to set a memory policy (bind, interleave, preferred or default) for a memory range. However, the use of such system calls is a complex task, since developers must deal with pointers, memory pages, sets of bytes and bit masks.

The second part of this support is a library named libnuma, which is a wrapper layer over the kernel system calls. The limited 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.

The numactl tool allows the user to set a memory policy for an application without changing the source code. However, the chosen policy will be applied over all application data (it is not possible to either express different access patterns or change the policy during the execution [12]). Additionally, the user must give as argument a list of nodes (memory banks and cpus/cores) that will be used, which is platform-dependent parameter.

2.4 Conclusion on Related Works

Most of the proposed solutions, presented in this section, do not avoid changes in application source code. They offer a limited set of memory policies (do not allow to express different memory pattern accesses) and they do not offer architecture abstraction (developer must know the target architecture). Moreover, they do not include optimizations for numerical scientific data (array data structures), which are intensively used in scientific numerical HPC applications.

inria-00421546, version 1 - 2 Oct 2009

inria-00421546, version 1 - 2 Oct 2009

3 Minas

Minas [13] is a framework that allows developers to manage memory affinity in an explicit or automatic way on large scale ccNUMA platforms. It is composed of three modules: Minas-MAi, Minas-MApp and numarch. Minas-MAi, which is a high level interface, is responsible for implementing the explicit NUMA-aware application tuning mechanism whereas the Minas-MApp preprocessor implements an automatic NUMA-aware application tuning. The last module, numarch, extracts all information about the target platform. This framework is efficient and portable, since it has good performance and provides architecture abstraction. Additionally, it has been designed to deal with NUMA penalties for numerical scientific HPC applications.

Minas differs from other memory affinity solutions [12, 9, 6] in at least four aspects. First of all, Minas offers code portability. Since numarch provides architecture abstraction, once the application source code is optimized for a specific ccNUMA platform, it can be used in another platform without any modifications. If the same memory affinity strategy fits both platforms, the performance gains will also be equivalent. Secondly, Minas is a flexible framework since it supports two different mechanisms to control memory affinity (explicit and automatic tuning). Thirdly, Minas is designed for array oriented applications, since this data structure usually represents the most important variables in kernels/computations. Finally, Minas implements several memory policies to deal with both regular applications (threads always access the same data set) and irregular applications (threads access different data during the computations).

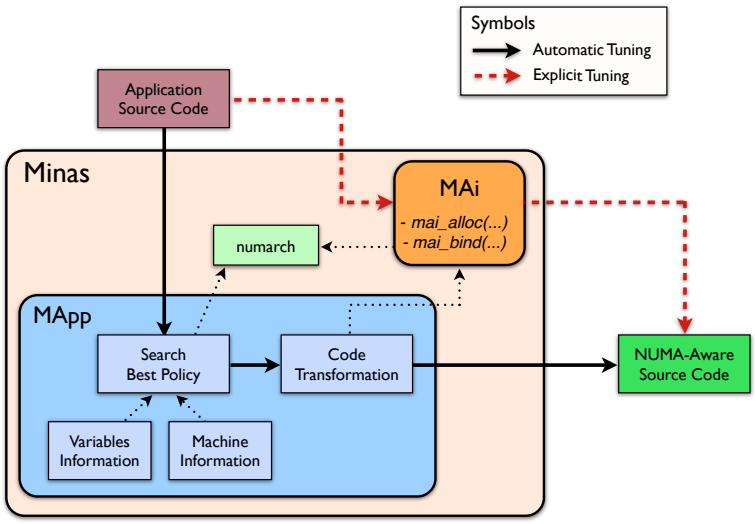


Figure 1: Overview of Minas.

Figure 1 shows a schema of Minas mechanisms to assure memory affinity. The original application source code can be modified by either using the explicit mechanism or the automatic one. The decision between automatic and explicit mechanisms depends on the developer's knowledge about the target application. One possible approach is to first use the automatic tuning mechanism and to verify whether the performance improvements are considered sufficient or not.

If the gains are not sufficient, developers can then explicitly modify (manual tuning) the application source code using Minas-MAi.

MAi is a user level interface that provides several memory policies (allocation and data placement) to explicitly manage memory affinity on numerical scientific applications. In order to use MAi interface, developers must only know characteristics of the application (memory access), since Minas can retrieve information about the target platform. More details about this interface are presented in section 3.1. MApp is a preprocessor that performs automated tuning of applications in order to minimize NUMA penalties. As such, developers do not need to manually change their source code as application code optimizations are done automatically using Minas knowledge of the underlying platform and application. Those optimizations are performed at compile time integrating MAi interface functions to application source code. More details about this preprocessor are presented in section 3.2.

The numarch module has an important role for both explicit and automatic mechanisms. In Minas-MAi, it is used to provide architecture abstraction for all memory policies. On the contrary, in Minas-MApp, it is used to consult all necessary information about the underlying platform in order to decide which policy will be applied during the code transformation.

The current version of Minas is implemented in C. Minas has been tested on different ccNUMA architectures (Intel, AMD and SGI) with Linux as operating system. Regarding languages and compilers, Minas supports C/C++, Intel C Compiler (ICC), GNU C Compiler (GCC) and Portland C Compiler (PGI). A Fortran support is currently underway.

3.1 MAi: Memory Affinity Interface

MAi[1] (Memory Affinity interface) is an API (Application Programming Interface) that provides a simple way to control memory affinity on application over ccNUMA platforms. It simplifies memory affinity management issues, since it provides simple and high level functions that can be called in the application source code to deal with data allocation and placement [14]. All MAi functions are array-oriented and they can be divided in three groups: allocation, memory policies and system functions. Allocation functions are responsible for allocating arrays (they are optimized for ccNUMA platforms). Memory policies functions are used to apply a specific memory policy for an array, allocating its memory pages on memory blocks. MAi has several memory policies that can be used to optimize memory access on ccNUMA platforms (latency and bandwidth optimization). System functions allows developers to collect and print system information such as memory blocks used by the memory policies, cpus/cores used during the application execution, memory blocks and statistics about page migration. Furthermore, MAi has a thread scheduling mechanism that are used with some memory policies to better assure memory affinity.

The most important group of functions of MAi is the memory policies, since it is responsible for assuring memory affinity. The interface implements eight memory policies, that have as memory affinity unit an array (Minas was designed for numerical scientific applications). The memory policies of MAi can be divided in three groups: bind, cyclic and random. The bind group is com-

[1] MAi can be download from http://mai.gforge.inria.fr/

inria-00421546, version 1 - 2 Oct 2009

posed of *bind_all* and *bind_block* memory policies, the cyclic one of *cyclic*, *cyclic_block*, *skew_mapp* and *prime_mapp*, and the random one of *random* and *random_block*. The main differences between these three groups are the memory blocks used, data distribution and thread scheduling. In MAi, data distribution can be performed using either individual elements of an array (element-by-element distribution) or an array block (blocks distribution). A block is a set of rows or columns, where the size can be specified by the user. If the user does not specify it, MAi will choose the block size automatically. The block size is computed considering the scheduling of the workload for the threads. This strategy is also applied for all MAi memory policies that use the concept of blocks.

In bind group, the distribution of an array is restricted to a set of memory blocks of the platform. *Bind_all* memory policy places all data in restricted memory block(s) specified by the user. If more than one memory block is specified, data will be placed in more memory blocks. However, this policy will use all available memory (physical) from the first memory block before using the next one. In *bind_block* memory policy, data is divided into blocks depending on the number of threads that will be used and where they are running. Due to this, blocks of data are placed closer to threads which will compute them.

In Figure 2 (a) and Figure 2 (b), we show a schema that represents how data distribution is done in *bind_all* and *bind_block* memory policy. A node n is composed of a memory block (Mn) and a set of processing units (to simplify the representation they were not shown). Bind memory policies were designed for applications that present a regular behavior. In such applications, each thread always accesses the same set of data and a static scheduling of the workload is used. Furthermore, bind policies optimize latency over ccNUMAs, since data is placed closer to the thread that uses it.

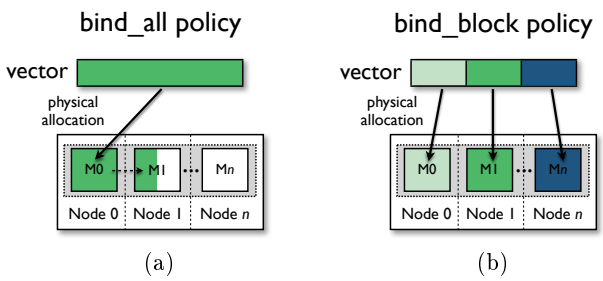


Figure 2: Bind memory policies.

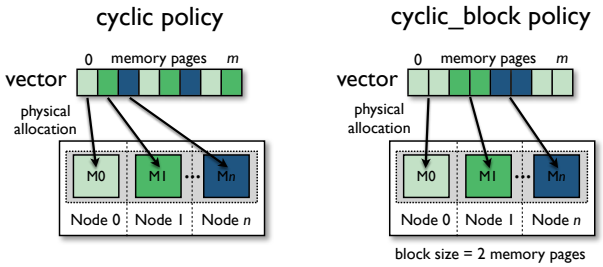
The cyclic group uses different round-robin strategies to place data on memory blocks. In both *cyclic* and *cyclic_block* policies, data is placed according to a linear round-robin distribution. However, *cyclic* uses a memory page per round (Figure 4 (a)), which has similar behavior of the interleave policy of the Linux NUMA support, whereas *cyclic_block* uses a block of memory pages (Figure 4 (b)).

The *skew_mapp* memory policy was proposed in [7] and it is a modification of round-robin policy that has linear page skew. In this policy, a page i is allocated on the node $(i + \lfloor i/M \rfloor + 1)\ mod\ M$, where M is the number of memory blocks (Figure 4 (a)). The *prime_mapp* policy was also proposed in [7] and uses a two-phase strategy. In the first phase, the policy places data using

inria-00421546, version 1 - 2 Oct 2009

cyclic policy on (P) virtual memory blocks, where P is a prime greater or equal to M (real number of memory blocks). In the second phase, the memory pages previously placed on virtual memory blocks are reordered and placed on real memory blocks also using the *cyclic* policy (Figure 4 (b)).

Cyclic memory policies can be used in applications with regular and irregular behavior (threads do not always access the same data). These memory policies spread data between the memory blocks minimizing concurrent accesses and increasing bandwidth. However, some scientific applications can still have contention problems with *cyclic* and *cyclic_ block*, since these policies make a linear distribution of memory pages (generally, power of 2) on the platform (it has power of 2 memory blocks). Thus, the proposal of *skew_ mapp* and *prime_ mapp* memory policies aims at reducing concurrent accesses for such applications [7].



(a) (b)

Figure 3: Cyclic memory policies

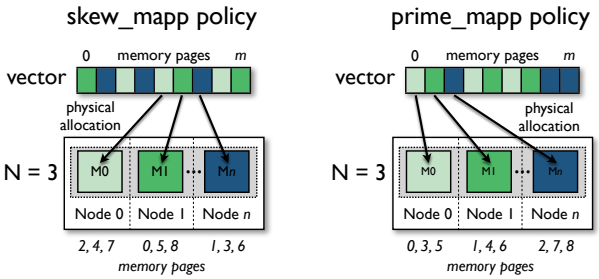
Finally, the last group of memory policies is *random*. In these memory policies, memory pages are placed randomly on ccNUMA nodes, using a random uniform distribution. The main goal of this memory policy is to increase bandwidth. Like other policies, different sizes of blocks can also be used.

One of the most important features of MAi is that it allows the developer to change the memory policy applied to an array during the application execution. This characteristic allows developers to express different patterns during the application execution. Additionally, MAi memory policies can be combined during the application execution to implement a new memory policy. Finally, any incorrect memory placement can be optimized through the use of MAi memory migration functions. The unit used for migration can be a set memory pages (automatically defined by MAi) or a set of rows/columns (specified by the user).

For bind memory policies, to better ensure memory affinity, both threads and memory must be considered in the solution. Due to this, MAi has a thread scheduling mechanism that is used for bind memory policies. The default thread scheduling policy is to fix them on processors/cores. Such strategy assures that threads will not migrate (less overhead with task migrations) and consequently, MAi will be able to perform a better data distribution and assure memory affinity. This thread scheduling considers the number of threads (T) and processors/cores (P) to decide how to fix threads. If $T \leq P$, one thread per processor/core strategy is chosen, which minimizes the memory contention problem on node, which is present in some NUMA platforms[2]. Memory contention hap-

[2]Bull Novascale Itanium 2 and SGI Altix Itanium 2 NUMA platforms.

inria-00421546, version 1 - 2 Oct 2009



(a) (b)

Figure 4: Cyclic memory policies

pens when several threads try to access the same memory block. Concurrent accesses in the same memory block can generate worse performance, since they must be serialized. The default scheduling strategy can be changed during the library initialization. The developer can then choose between using the operating system thread scheduling or defining his own threads and processors/cores mapping.

3.2 MApp: Memory Affinity Preprocessor

MApp (Memory Affinity preprocessor) is a preprocessor that provides a transparent control of memory affinity for numerical scientific HPC applications over ccNUMA platforms. MApp performs optimizations in the application source code considering the application variables (shared arrays) and platform characteristics at compile time. Its main characteristics are its simplicity of use (automatic NUMA-aware tuning, no manual modifications) and its platform and compiler independence.

The code transformation process is divided into four steps. Firstly, it scans the input file (application source code) to obtain information about variables. During the scanning process, MApp searches for shared static arrays that are considered large by Minas (eligible arrays). An eligible array is considered large if its size is equal or greater than the size of the highest level cache of the platform. Secondly, it fetches the platform characteristics, retrieving information from the numarch module (NUMA factor[3], nodes, cpus/cores, interconnection network and memory subsystem). During the third step, it chooses a suitable memory policy for each array. Finally, the code transformation is performed by changing the static arrays declaration and including Minas-MAi specific functions for allocation and data placement.

The most important step of MApp automatic tuning process is the strategy used to decide which memory policy will be applied for an application. Based on empirical data from our previous works and experiments [15, 14, 16], we have designed an heuristic responsible for deciding which memory policy would be the most effective considering the underlying ccNUMA characteristics. On platforms with a high number of interconnections between nodes (e.g., fat-tree and hypercube) and small NUMA factor, the heuristic will apply cyclic memory policies, since such platforms usually present memory contention problems. On

[3]NUMA factor is the ratio between remote latency and local latency.

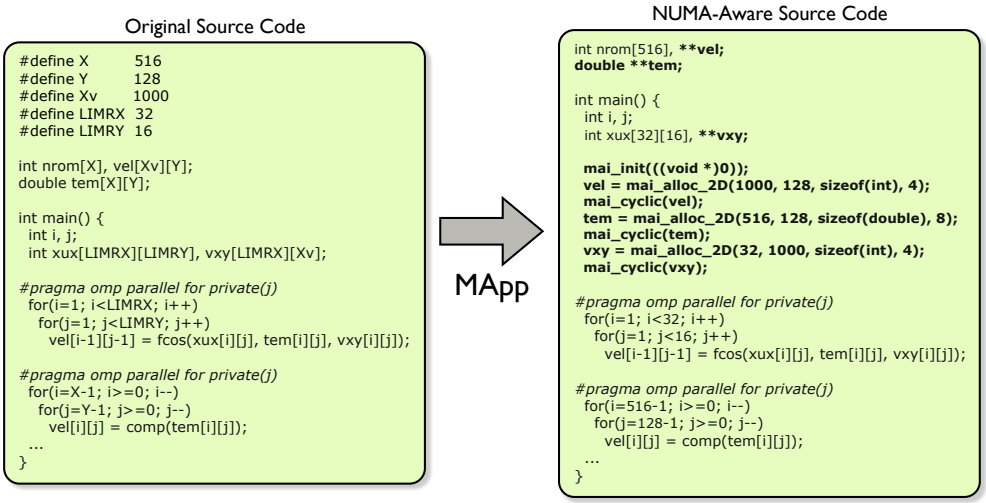
inria-00421546, version 1 - 2 Oct 2009

Program 1 An heuristic to select a memory policy.

begin
arch:=*get_numarch*();
do **if** arch.numaf >= 2 **then**
do **if** *!complexnet*(arch.net) **then**
mpolicy:=bind; **exit** **fi**
else
mpolicy:=bind;
block:=*verifydata*(rows,col);
exit **fi**
end

the contrary, on platforms with low number of interconnections and high NUMA factor, the heuristic will opt for *bind_block* memory policies (Program 1).

Figure 3.2 shows a simple example of a code transformation generated by MApp. This example is a parallel code (C with openMP) that performs some operations in four arrays. However, as we can observe, MApp only applied memory policies for three of them (eligible arrays). Small variables such as *i*,*j* and *xux* will probably fit in cache so MApp will not interfere on compiler decisions (allocation and placement of variables). In this example, the target ccNUMA platform has a small NUMA factor (remote latency is low) and a bandwidth problem for interconnection between nodes. Thus, on such a platform, optimizing memory accesses considering bandwidth instead of latency is important. Due to this, MApp has decided to spread memory pages of *vel*, *vxy* and *tem* with *cyclic* memory policy in order to optimize bandwidth.



captionExample of MApp source code transformation.

3.3 Numarch

Numarch is a module of Minas that extracts information from the underlying ccNUMA platform. The information retrieved by numarch is used by MAi and

inria-00421546, version 1 - 2 Oct 2009

MApp to provide architecture abstraction and assure memory affinity. Its main characteristics are its simplicity of use (interface with high level functions) and portability (support for different platforms).

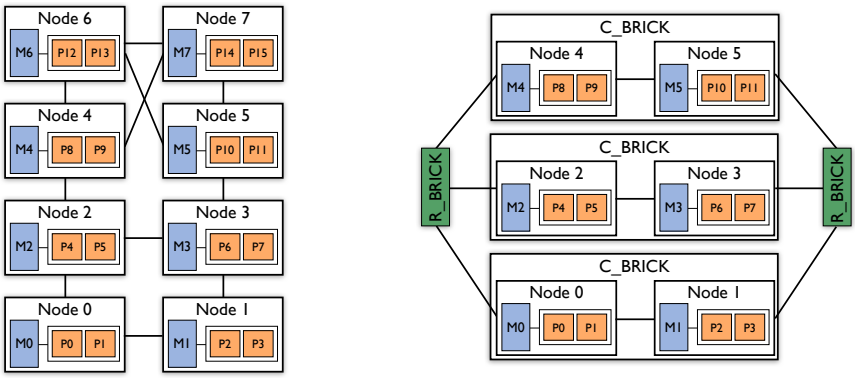
This module extracts information about the interconnection network (number of links and bandwidth), memory access costs (NUMA factor) and architecture (number of nodes and cpus/cores). To retrieve such information, numarch parses the file system of the operating system. Numarch is implemented part in C and part in shell script.

4 Minas Evaluation

In this section, we present the evaluation of Minas compared with the other three memory affinity solutions for Linux based platforms. We first describe the two ccNUMA platforms used in our experiments. Then, we describe the four numerical scientific applications (NAS Parallel Benchmarks [17], ICTM [16] and Ondes 3D [18]) and their main characteristics. Finally, we present architecture abstraction and performance results.

4.1 ccNUMA Platforms

Our experiments have been carried out on two ccNUMA platforms. The first platform is an eight dual core AMD Opteron 2.2 GHz. It is organized in eight nodes of two processors with 2 MB of shared cache memory for each node. It has a total of 32 GB of main memory (4 GB of local memory). Each node has three connections (HyperTransport [19]) which are used to link with other nodes (except nodes zero and one). These connections give different memory latencies for remote access by nodes of the platform. The NUMA factor on this platform varies from **1.2** to **1.5**. The compiler that has been used for the OpenMP code compilation was the GCC (GNU C Compiler). A schematic representation of this machine is given in Figure 5 (a). We have chosen to use the name **Opteron** for this platform.



(a) (b)

Figure 5: NUMA Platforms: (a) Opteron (b) SGI.

inria-00421546, version 1 - 2 Oct 2009

The second ccNUMA platform is a SGI Altix 350 with twelve Itanium 2 processors of 1.5 GHz and 4 MB of shared cache memory each. It is organized in six nodes of two processors with a total of 24 GB of main memory (4 GB of local memory). Each node has two connections (NUMAlink switch [20]) which are used to link with other nodes. The NUMA factor for this platform varies from **1.2** to **1.3**. The compiler that has been used for the OpenMP code compilation was the ICC (version 9.0). A schematic representation of this machine is given in Figure 5 (b). We have chosen to use the name **SGI** to make reference to this platform.

The operating system that has been used for both platforms is Linux 64-bits version with support for NUMA architecture (system calls, NUMA API and user tool *numactl*).

4.2 Applications

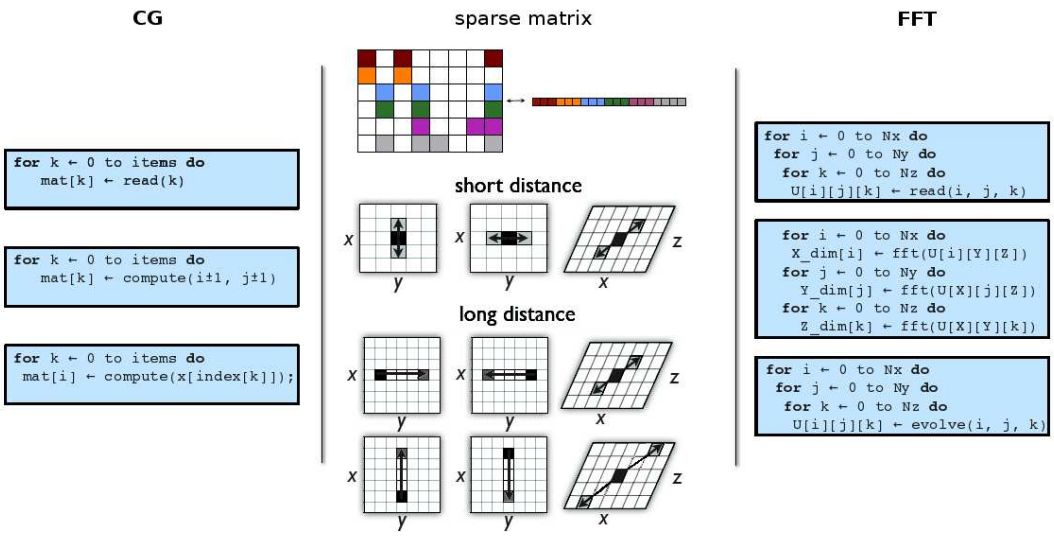
In this section, we present the four applications we have used to evaluate Minas performance. We have selected three applications, two from NAS Parallel Benchmarks (NPB's) and two Geophysics applications. In this work, we have chosen the kernels Conjugate Gradient (CG) and Fast Fourier Transform (FFT) from NPB's[4]. The Geophysics applications were the Interval Categorizer Tessellation Model (ICTM) and the Simulation of Seismic Wave Propagation (Ondes 3D). Such applications have been selected because they represented important numerical scientific problems. They also are memory-bound and they have regular/irregular memory access patterns. All applications have been implemented in C and they have been parallelized using OpenMP.

NAS Parallel Benchmarks - CG kernel. NAS Parallel Benchmarks is a well-known benchmark derived from Computational Fluid Dynamics (CFD) codes and it is composed of applications and kernels [17]. CG is a kernel that uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel tests unstructured vector computations and communications. The computations are basically sparse-matrix vector multiplication, reduction sums and several vector operations performed in parallel. It uses a sparse-matrix vector with randomly generated locations of entries which gives a large amount of cache misses. The input parameter of this kernel is the size of the array that is used for the computation. In this work, we have used an array of size 75000 (class B, 6.7 Gbytes). Figure 6 shows the algorithm (Figure 6 (a)) and its memory accesses patterns (Figure 6 (b)). Basically, CG uses loops with random and long distance memory accesses during the computation phases over sparse matrices represented with vectors.

NAS Parallel Benchmarks - FFT kernel. FFT is a kernel that computes the fast transform of Fourier for three dimensional systems. The application works with complex numbers that are represented by data structures. There are three main steps in the FFT computation and data are shared just in the second step. The computation is done in one direction by step and each thread

[4]Results for other NPB's kernels and applicatins can be found in Minas project homepage [13].

inria-00421546, version 1 - 2 Oct 2009



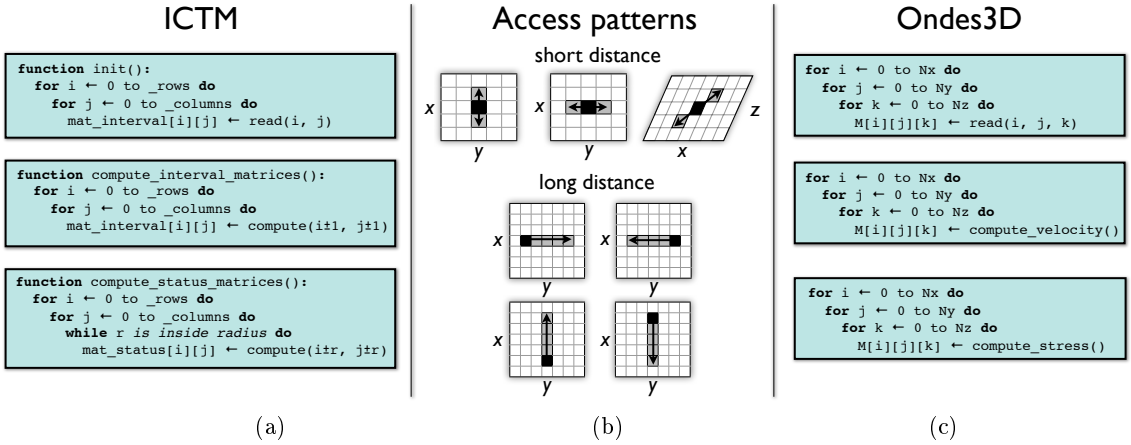
(a) (b) (c)

Figure 6: Access patterns: CG and FFT.

computes nz-planes. Figure 6 (c) presents a schema of the application. FFT memory access patterns (short and long in X, Y and Z direction) are presented in Figure 6 (b) . In our experiments, we have used $512x256x256$ matrices (class B, 1.25 Gbytes).

ICTM: Interval Categorizer Tessellation Model. ICTM is a multi-layered tessellation model for the categorization of geographic regions considering several different characteristics (relief, vegetation, climate, etc.). The number of characteristics that should be studied 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, in which the information is given in certain points referenced by their latitude and longitude coordinates. The geographic region is represented by a initial 2-D matrix of the total area into sufficiently small rectangular subareas. In order to categorize the regions of each layer, ICTM executes sequential phases. Each phase accesses specific matrices that have previously been computed and generates a new 2-D matrix as a result of the computation. Depending on the phase, the access pattern to other matrices can either be regular or irregular. Since the categorization of extremely large regions has a high computational cost, a parallel solution for ccNUMA platforms has been proposed in [16]. In this paper, we have carried out experiments using 6700x6700 matrices (2 Gbytes of data) and a radius of size 40 (number of neighbors to be analysed by status matrix phase). As shown in Figure 7 (a), the algorithm basically uses nested loops with short and long distance memory accesses (Figure 7 (b)) during the computation phases.

inria-00421546, version 1 - 2 Oct 2009



(a) (b) (c)

Figure 7: Access patterns: ICTM and Ondes 3D.

Ondes 3D: Simulation of Seismic Wave Propagation. Ondes 3D is an application that simulates seismic wave propagation in three dimensional geological media based on finite-difference discretization. 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 simulation 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. Ondes 3D has three main steps: data allocation, data initialization and propagation calculus (composed by two calculus loops). During the first two steps, the three dimensional arrays are dynamically allocated and initialized (400x400x400, approximately 4.6 Gbytes of memory). During the last step, the two calculus loops compute velocity and stress of the seismic wave propagation. In all three steps, the three dimensional arrays are accessed in a regular way (same data access pattern) [18, 15]. Figure 7 (c) presents a schema of the application with its three steps. On contrary to ICTM, Ondes 3D has only short distance memory accesses, as presented in Figure 7 (b).

4.3 Architecture Abstraction

In this section we present the evaluation of architecture abstraction of Minas using two applications (CG and ICTM) and platforms described in previous sections. For each application and platform, we have compared Minas with *numactl* and *libnuma*.

Regarding the explicit memory affinity solutions, we have changed the applications execution parameters (*numactl*) and their source codes (Minas-MAi and *libnuma*). The *numactl* tool avoids source code modifications but demands some parameters for the application execution command line. In order to use Minas-MAi and *libnuma*, the developer must add specific data allocation and placement functions in the source code.

inria-00421546, version 1 - 2 Oct 2009

CG Kernel – Opteron and SGI

```
#4threads - Opteron
numamactl --interleave=0,1,2,3 --physcpubind=0,2,4,6  ./cg.B

#2threads - SGI
numactl --interleave=0,1 --cpubind=0-3  ./cg.B
```

Figure 8: CG Kernel with numactl.

The explicit solution *numactl* does not require source code changes. However, we had to change the execution command line of all applications to specify which memory policy should have been applied as well as the nodes and cpus lists. Figures 8 and 9 show the command line that has been used to execute CG and ICTM. Contrary to Minas-MAi, *numactl* do not allow architecture abstraction, since the nodes and cpus lists are explicit passed as parameters to the tool.

ICTM – Opteron and SGI

```
#4threads - Opteron
numamactl --interleave=0,1,2,3 --physcpubind=0,2,4,6  ./ictm 6700 40 4

#2threads - SGI
numactl --interleave=0,1 --cpubind=0-3  ./ictm 6700 40 2
```

Figure 9: ICTM with numactl.

To apply Minas-MAi in CG and ICTM applications, we have selected three different memory policies (*cyclic*, *skew_mapp* and *bind_block*). The first two memory policies are ideal for irregular memory accesses (CG and ICTM) over ccNUMA platforms that have a small NUMA factor, since they spread data among nodes. The latter memory policy is suitable for regular memory accesses where threads always access the same data set. Since *libnuma* has a limited set of memory policies, we have used two strategies. The interleave policy (similar behavior of Minas-MAi cyclic policy) has been applied for irregular memory accesses whereas the *numa_tonode_memory()* function has been used for regular ones. In Figure 10 (a), (b), (c) and (d), we present a snippet of ICTM and CG applications coded with Minas-MAi and *libnuma*. As we can observe, the implementation with Minas-MAi is simpler than the *libnuma* one. The main difference between Minas-MAi and *libnuma* is that the first one can gather the platform characteristics, so that no explicit configuration is needed to apply memory policies.

inria-00421546, version 1 - 2 Oct 2009

Minas-MAi

```
//initialize MAi
mai_init();

//allocate data
_intY=(float**)mai_alloc_2D(_rows,_columns,
                 sizeof(float),FLOAT);
_monoRight=(int**)mai_alloc_2D(_rows,
            _colum ns, sizeof(int),INT);

//select memory policy
mai_cyclic_rows(_intY,_rows/_numThreads);
mai_cyclic(_monoRight);
```

(a)

Libnuma

```
//set nodes mask
nodemask_zero(&_nodes);
nodemask_set(&_nodes,2);
nodemask_set(&_nodes,3);
numa_set_interleave_mask(&_nodes);

//allocate with interleave policy
_intY=(float**)numa_alloc_interleaved_subset
               (sizeof(float *)*_rows,&_nodes);
_monoRight=(int**)numa_alloc_interleaved_subset
               (sizeof(int *)*_rows,&_nodes);

_intY[0]=(float*)numa_alloc_interleaved_subset
        ((sizeof(float)*_rows*_columns),&_nodes);
 for(i=1;i<_rows;i++)
   _intY[i]= _intY[i-1]+_columns;

_monoRight[0]=(int*)numa_alloc_interleaved_subset
           (sizeof(int)*_rows *_columns,&_nodes);
 for(i=1;i<_rows;i++)
   _monoRight[i]= _monoRight[i-1]+_columns;
```

(b)

Minas-MAi

```
//initialize MAi
mai_init();

//allocate data
iv=mai_alloc_1D(2*NA+1+1,sizeof(int),INT);
aelt=mai_alloc_1D(NZ+1,sizeof(double),
                  DOUBLE);

//select memory policy
mai_cyclic(iv);
mai_skew_mapp(aelt);
```

(c)

Libnuma

```
//set nodes mask
nodemask_zero(&_nodes);
nodemask_set(&_nodes,2);
nodemask_set(&_nodes,3);
numa_set_interleave_mask(&_nodes);

//allocate with interleave policy
iv=numa_alloc_interleaved((2*NA+1+1)*
                          sizeof(int));
aelt=numa_alloc_interleaved((NZ+1)*
                           sizeof(double));
```

(d)

Figure 10: CG Kernel and ICTM with Minas-MAi and libnuma.

4.4 Performance Evaluation

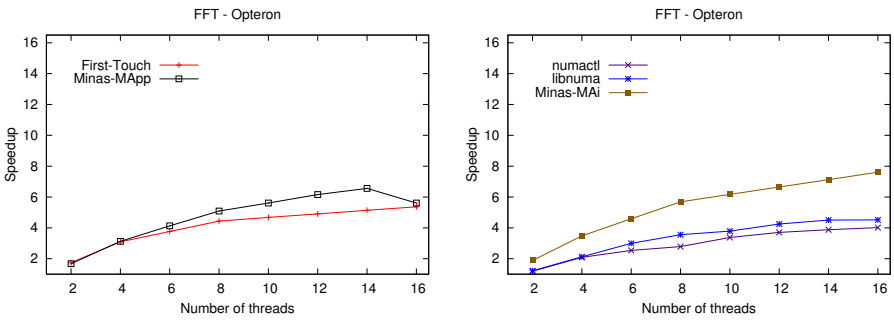
In this section we present the performance evaluation of Minas using the applications and platforms described in previous sections. For each application and platform, we have carried out series of experiments using Minas and three Linux solutions (*first-touch* policy, *numactl* and *libnuma*).

The results have been obtained through the average of several executions varying the number of threads from 2 to the maximum number of cpus/cores of each platform. These results have presented a low standard deviation, since all experiments have been done with exclusive access to the ccNUMA machines.

inria-00421546, version 1 - 2 Oct 2009

Our results are organized by application (FFT, CG, ICTM and Ondes 3D). For each application, except for FFT, we have divided the results into two groups according to the memory affinity management (automatic: *First-Touch* and Minas-MApp; explicit: Minas-MAi, *numactl* and *libnuma*). In this work, for FFT, we only present results with Opteron platform.

Figure 11 shows the speedups for FFT on Opteron platform with the automatic (Figure 11 (a)) and the explicit (Figure 11 (b)) memory affinity solutions. As it can be observed, Minas has outperformed all other memory affinity solutions.



(a) (b)

Figure 11: Performance of FFT on the Opteron Platform (a) Automatic (b) Explicit.

As we can observe in Figure 11 (a), Minas-MApp have obtained better results than *first_touch*. Considering the characteristics of Opteron platform, Minas-MApp heuristic have chosen *cyclic* as memory policy to be applied in the eligible arrays of FFT. Such platform has a small NUMA factor and bandwidth optimizations are important. Additionally, FFT is an irregular application in which three dimensional arrays are accessed in a non linear way. On general, *first-touch*, have not presented good results. As discussed earlier, this memory policy optimizes latency and considering this platform and application *first-touch* is not a efficient choice. We can also observe that the results with Minas-MApp and *first-touch* have been similar for two, four and sixteen threads. When a small number of threads is used memory contention is not high, thus different memory policies may have the similar performance on platforms with small NUMA factor (remote access costs are not high). *First-touch* and Minas-MApp have obtained similar results with sixteen threads because in this case, both memory policies spread memory pages over all memory banks of the platform.

Considering the explicit solutions applied to FFT, Minas-MAi have presented better results when compared with *libnuma* and *numactl*. In this application, we have used the *prime_mapp* memory policy of Minas-MAi. Such policy aims at providing a non-uniform distribution of memory pages among the ccNUMA nodes. Due to this fact, it spreads memory pages better than *cyclic* memory policies, since it avoids any patterns during data distribution. On *libnuma*

inria-00421546, version 1 - 2 Oct 2009

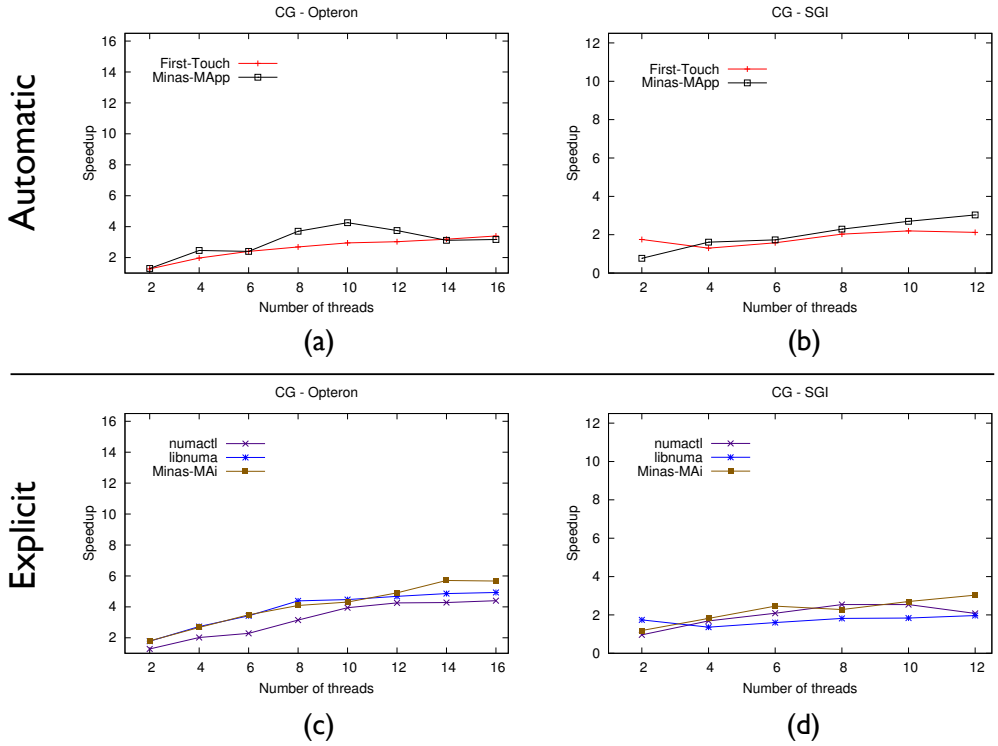


Figure 12: Performance of CG on Opteron and SGI platforms.

and *numactl*, we have implemented FFT using interleave memory policy, since these solutions do not have *prime_mapp* memory policy. They presented similar results for FFT, but *libnuma* improvement gains have been greater than *numactl*. *Libnuma* allows developers to apply memory policies to a specific memory range whereas *numactl* applies a memory policy for all application data.

In Figure 12, we present the speedups obtained with CG on the two ccNUMA platforms using the automatic and explicit solutions. As we can observe, Minas has performed well on both platforms. Considering the automatic solutions (Figure 12 (a) and (b)), we have noticed that Minas-MApp generally presented better results (11% better than *first_touch*, on average). Even though the results of *libnuma* and Minas-MAi were very similar (Figure 12 (c) and (d)), Minas-MAi has presented higher performance gains (on average, 3% on Opteron and 24% on SGI). We can also notice that *numactl* presented the worst results within the explicit solutions group.

Concerning the CG results with automatic solutions, *first_touch* generally has resulted in worse performance when compared to Minas-MApp. This policy is not suited to irregular applications since it optimizes latency instead of reducing memory contention. This optimization results in several memory accesses on the same memory banks. In this case, considering the platforms network interconnections, the heuristic implemented in Minas-MApp has selected *cyclic*

inria-00421546, version 1 - 2 Oct 2009

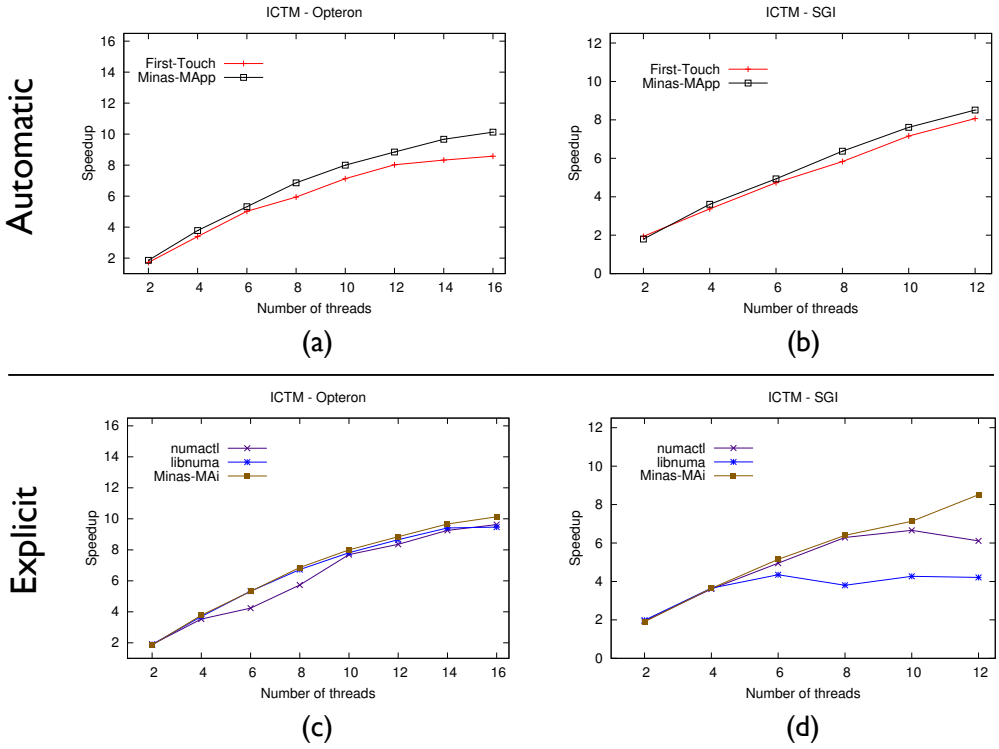


Figure 13: Performance of ICTM on Opteron and SGI platforms.

as the best memory policy for both platforms, which has minimized the memory contention problem.

Regarding the explicit solutions, in CG with Minas-MAi, we have used *cyclic* and *bind_ block* memory policies on Opteron and *skew_ mapp* memory policy on SGI platform. Since Opteron has a low NUMA factor and a simple interconnection network, we have applied *cyclic* for arrays that are accessed irregularly, whereas *bind_ block* has been applied for those accessed regularly. Thus, we can both optimize bandwidth and reduce memory contention. On the contrary, SGI has a complex network topology (fat-tree) and bandwidth optimization is also an important concern. Because of that, *skew_ mapp* has been used in CG kernel on SGI platform, since it distributes memory pages in a non-uniform way, reducing the number of concurrent accesses on nodes. On both platforms, *numactl* has shown less performance improvements in comparison with other explicit solutions. Since it applies a memory policy for all application data, we cannot express memory access patterns.

Figure 13 shows the speedups for ICTM on Opteron and SGI platforms with the automatic (Figure 13 (a) and (b)) and the explicit (Figure 13 (c) and (d)) memory affinity solutions. As it can be observed, Minas has outperformed all other memory affinity solutions on both platforms.

inria-00421546, version 1 - 2 Oct 2009

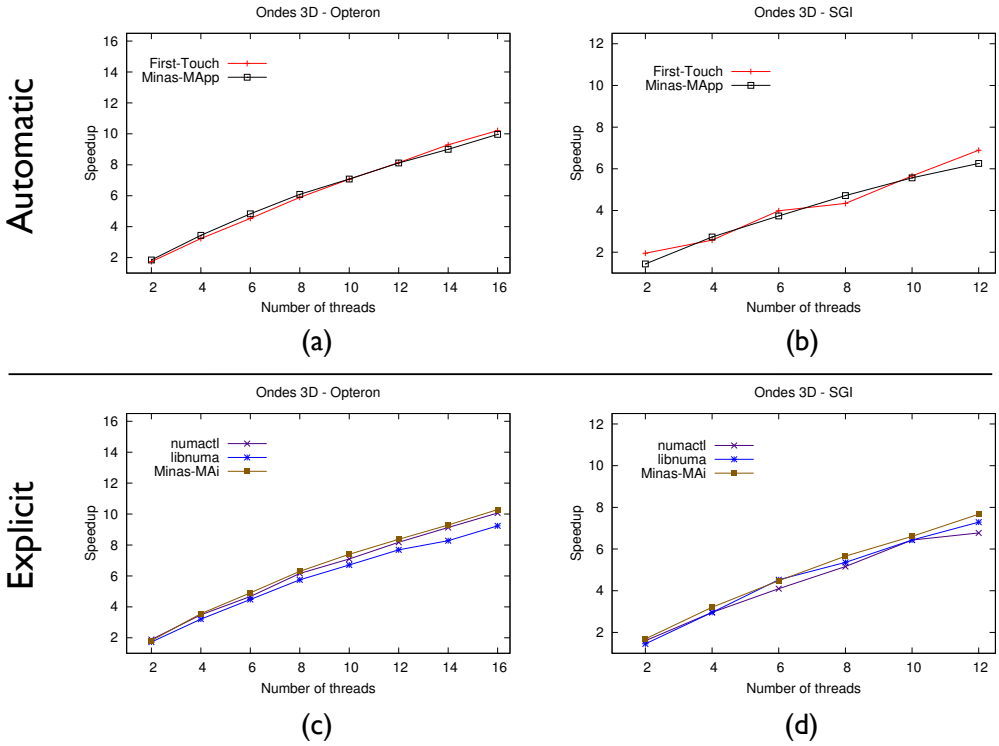


Figure 14: Performance of Ondes 3D on Opteron and SGI platforms.

Considering the automatic solutions applied to ICTM, Minas-MApp has presented satisfactory results on both platforms (Figure 13 (a) and (b)). Minas-MApp has chosen *cyclic* memory policy to control data allocation and placement on both platforms. This chosen policy has resulted in better performance gains than *first_touch* (on average, 10% Opteron and 8% on SGI). After a deep analysis of these results, we have concluded that *first_touch* policy has generated more remote accesses. This behavior has had an impact on its gains on Opteron, which has a higher NUMA factor.

The explicit solutions have presented different behaviors depending on the platform (Figure 13 (c) and (d)). On Opteron, the Minas-MAi *cyclic* memory policy has presented the best results. However, there is not a significant difference between Minas-MAi and other explicit solutions (*libnuma* and *numactl*). This can be explained by the fact that *libnuma* and *numactl* also offer a similar policy, named interleave. We believe that the slight performance gains of Minas-MAi are due to the array optimizations (specialized allocation functions and false sharing reduction). On SGI, the three solutions have presented different performance gains. We believe that Minas-MAi has also presented a better performance thanks to the array optimization included in allocation functions and memory policies. However, the platform network interconnection characteristics could have had an impact on data distribution and accesses to memory

inria-00421546, version 1 - 2 Oct 2009

on ICTM (irregular accesses). For this application and this platform, more experiments must be performed to better comprehend these results.

In Figure 14, we show the speedups for Ondes 3D application on Opteron and SGI platforms with the automatic (Figure 14 (a) and (b)) and the explicit (Figure 14 (c) and (d)) memory affinity solutions. On both platforms, Ondes 3D application with Minas has presented better performance gains than the other solutions for memory affinity control.

The results obtained with automatic solutions in Ondes 3D have shown that *first_ touch* had an overall performance gains compared to Minas-MApp. The Minas-MApp heuristic has chosen *cyclic* as the best policy according to the platform characteristics. However, as discussed before, the best policy for this application on such platforms is Minas-MAi *bind_ block*. Since, *first_ touch* and *bind_ block* have similar behavior, their results are expected to be equivalent or superior to the Minas-MApp choice.

Finally, the results with explicit solutions in Ondes 3D (Figure 14 (c) and (d)) have shown that *libnuma* and *numactl* have had a worse performance than Minas. Since this application has a regular memory access, it is important to maintain both thread and their data as close as possible. In order to do so, data should be divided among NUMA nodes and threads should be fixed on cores/cpus of such nodes. This strategy can be achieved by either Minas-MAi or *libnuma*. However, *libnuma* demands considerable codification efforts, since developers must implement all data distribution algorithm and thread scheduling. Additionally, the same solution may not work on platforms with different architecture characteristics. In contrast with *libnuma*, Minas-MAi provides a specific policy for this purpose which is called *bind_ block*. This policy automatically fixes threads and distributes data among the NUMA nodes (architecture abstraction). Thus, no source changes are needed when the same solution is applied on different platforms. *Numactl* is the less flexible of all explicit solutions and it does not provide such data distribution strategy (in this case we have used the interleave policy).

Table 1: Impact of Minas automatic tuning (Minas-MApp) mechanism.

	FFT	CG	ICTM	Ondes 3D
Opteron	[4%; 12%]	[1%; 25%]	[0%; 0%]	[0%; 3%]
SGI	-	[0%; 21%]	[0%; 0,5%]	[10%; 13%]

In Table 1, we present the minimum and maximum performance losses of Minas automatic tuning mechanism (Minas-MApp) in comparison with Minas explicit tuning mechanism (Minas-MAi) for each application and platform. We can notice that in some cases, Minas-MApp had an insignificant impact in terms of performance in relation with Minas-MAi (ICTM on both platforms and Ondes 3D on Opteron). However, according to our experiments, the performance loss may be important (up to 25%). Thus, Minas-MApp could be a possible solution when developers do not choose to explicitly modify the application source code.

inria-00421546, version 1 - 2 Oct 2009

5 Conclusion and Future Work

In this paper, we have focused our work on Minas, a memory affinity management software to deal with memory placement on ccNUMA platforms for scientific HPC applications based on arrays. We also have presented its design, its approaches (MAi and MApp), its main functionalities, its implementation details and advantages (simplicity, efficiency, and portability). In order to evaluate its performance, we have carried out experiments over two ccNUMA platforms using CG and FFT form NPB's and two Geophysics applications.

Our experiments show that Minas performance improvement up to 44% in relation to the *first-touch* (CG kernel over opteron platform). Gains with both Minas approaches were observed for all applications over the two ccNUMA platforms used in our experiments. For these experiments, Minas presented better results than *first_touch* policy, *numactl* and *libnuma*.

Future work on Minas includes providing dynamic memory policies (next-touch, adaptive policies, etc), support for Fortran code and other threads libraries (Intel Thread Building Blocks, Posix Threads etc) on Minas preprocessor, defining hierarchical tiles for 3D/4D arrays, design and implementation of a runtime for OpenMP under GCC with implicit memory affinity control.

Acknowledgment

This research was supported by the French ANR under grant NUMASIS ANR-05-CIGC and CAPES (Brazil) under grant 4874-06-4.

References

[1] T. Mu, J. Tao, M. Schulz, and S. A. McKee, "Interactive Locality Optimization on NUMA Architectures," in *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*. New York, NY, USA: ACM, 2003, pp. 133–ff.

[2] J. Marathe and F. 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*. New York, NY, USA: ACM, 2006, pp. 90–99. [Online]. Available: http://portal.acm.org/citation.cfm?id=1122987

[3] A. Carissimi, F. Dupros, J.-F. Mehaut, and R. V. Polanczyk, "Aspectos de Programação Paralela em arquiteturas NUMA," in *VIII Workshop em Sistemas Computacionais de Alto Desempenho*, 2007.

[4] H. Löf and S. 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*. New York, NY, USA: ACM, 2005, pp. 387–392. [Online]. Available: http://portal.acm.org/citation.cfm?id=1088149.1088201

[5] C. Terboven, D. A. Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and Thread Affinity in OpenMP Programs," in *MAW '08:*

inria-00421546, version 1 - 2 Oct 2009

Proceedings of the 2008 workshop on Memory access on future processors. New York, NY, USA: ACM, 2008, pp. 377–384. [Online]. Available: http://dx.doi.org/10.1145/1366219.1366222

[6] B. Goglin and N. Furmento, "Enabling High-Performance Memory Migration for Multithreaded Applications on Linux," in *MTAAP'09: Workshop on Multithreaded Architectures and Applications, held in conjunction with IPDPS 2009*, IEEE, Ed., Rome Italie, 2009. [Online]. Available: http://hal.inria.fr/inria-00358172/en/

[7] R. Iyer, H. Wang, and L. Bhuyan, "Design and Analysis of Static Memory Management Policies for CC-NUMA Multiprocessors," College Station, TX, USA, Tech. Rep., 1998.

[8] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé, "User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors," in *ICPP '00: Proceedings of the 2000 International Conference on Parallel Processing*, 2000, pp. 95–104.

[9] A. Joseph, J. Pete, and R. Alistair, "Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport," 2006, pp. 338–352. [Online]. Available: http://dx.doi.org/10.1007/11945918_35

[10] D. S. Nikolopoulos, E. Artiaga, E. Ayguadé, and J. Labarta, "Exploiting Memory Affinity in OpenMP Through Schedule Reuse," *SIGARCH Computer Architecture News*, vol. 29, no. 5, pp. 49–55, 2001.

[11] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner, "Extending OpenMP for NUMA Machines," in *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Dallas, Texas, USA, 2000.

[12] A. Kleen, "A NUMA API for Linux," Tech. Rep. Novell-4621437, April 2005. [Online]. Available: http://whitepapers.zdnet.co.uk/0,1000000651,260150330p,00.htm

[13] C. P. Ribeiro and J.-F. Méhaut, "Minas Project - Memory affInity maNAgement System," 2009. [Online]. Available: http://pousa.christiane.googlepages.com/Minas

[14] C. Pousa, M. Castro, L. G. Fernandes, A. Carissimi, and J.-F. Méhaut, "Memory Affinity for Hierarchical Shared Memory Multiprocessors," in *21st International Symposium on Computer Architecture and High Performance Computing - SBAC-PAD (to appear)*. São Paulo, Brazil: IEEE, 2009.

[15] F. Dupros, C. Pousa, A. Carissimi, and J.-F. Méhaut, "Parallel Simulations of Seismic Wave Propagation on NUMA Architectures," in *ParCo'09: International Conference on Parallel Computing (to appear)*, Lyon, France, 2009.

inria-00421546, version 1 - 2 Oct 2009

[16] M. Castro, L. G. Fernandes, C. Pousa, J.-F. Méhaut, and M. S. de Aguiar, "NUMA-ICTM: A Parallel Version of ICTM Exploiting Memory Placement Strategies for NUMA Machines," in *PDSEC '09: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium - IPDPS*. Rome, Italy: IEEE Computer Society, 2009.

[17] J. Y. Haoqiang Jin, Michael Frumkin, "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance," NAS System Division - NASA Ames Research Center, Tech. Rep. 99-011/1999, 1999. [Online]. Available: https://www.nas.nasa.gov/Research/Reports/Techreports/1999/PDF/nas-99-011.pdf

[18] F. Dupros, H. Aochi, A. Ducellier, D. Komatitsch, and J. 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*, Sao Paulo, Brazil, 2008, pp. 253–260.

[19] AMD, "Advanced Micro Devices - AMD Opteron," 2009. [Online]. Available: http://www.amd.com

[20] SGI, "SGI NUMAlink Interconnect Fabric," 2009. [Online]. Available: http://www.sgi.com/products/servers/altix/\numalink.html

inria-00421546, version 1 - 2 Oct 2009

inria-00421546, version 1 - 2 Oct 2009

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399