# Improving Memory Affinity of Geophysics Applications on NUMA platforms Using Minas

Christiane Pousa Ribeiro[1] and Márcio Bastos Castro[1] and Jean-François Méhaut[1] and Alexandre Carissimi[2]

[1] University of Grenoble - LIG Laboratory
INRIA Mescal Research Team
Grenoble, France
(pousa, bastosca, Jean-Francois.Mehaut)@imag.fr
[2] Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
asc@inf.ufrgs.br

**Abstract.** On numerical scientific High Performance Computing (HPC), Non-Uniform Memory Access (NUMA) platforms are now commonplace. On such platforms, the memory affinity management remains an important concern in order to overcome the memory wall problem. Prior solutions have presented some drawbacks such as machine dependency and a limited set of memory policies. This paper introduces Minas, a framework which provides either explicit or automatic memory affinity management with architecture abstraction for ccNUMAs. We evaluate our solution on two ccNUMA platforms using two geophysics parallel applications. The results show some performance improvements in comparison with other solutions available for Linux.

## 1 Introduction

The increasing number of cores per processor and the efforts to overcome the hardware limitations 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 may vary depending on the distance between cpus and memory banks. The effects of this asymmetry can be reduced by optimizing memory affinity [1, 2].

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 on user and kernel space. However, such solutions present some drawbacks, such as: platform dependency (developers must have prior knowledge of the target architecture), they do not address different memory

accesses and they do not include optimizations for numerical scientific data (i.e., array data structures) [1–3].

To overcome these issues, our research have led to a new solution named 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. In this paper, we evaluate its portability and efficiency by performing experiments with two Geophysics applications on two ccNUMA platforms. The results are compared with Linux solutions for ccNUMAs (*first-touch*, *numactl* and *libnuma*).

This paper is organized as follows: first, we discuss the related work (Section 2). After presenting the Minas design, its characteristics and implementation details (Section 3), we will show its performance evaluation (Section 4). We will then give a brief conclusion and present our future work (Section 5).

## 2 Related Work

In order to guarantee memory affinity and thus achieve better performance, developers usually spend significant time optimizing data allocation and placement on applications and ccNUMA platforms. As a consequence, research groups have studied different ways to simplify memory affinity management on such platforms using Linux [2]. Two approaches have been proposed for the Linux operating system, the explicit approach (libraries, interfaces and tools) and the automatic approach (memory policies in user or kernel spaces) [3–6].

On the Linux operating system, the explicit approach is a basic support to manage memory affinity on ccNUMAs which is composed of 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. A memory policy is responsible for placing memory pages on physical memory banks of the machine. 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. The last part, the *numactl* tool, allows the user to set a memory policy for an application without changing the source code. However, the chosen policy is applied over all application data (it is not possible to either express different access patterns or change the policy during the execution [3]). Additionally, providing a list of nodes (memory banks and cpus/cores), that are platform-dependent parameters, is mandatory when using this tool.

The automatic approach is based on the use of memory policies and it is the simplest way to deal with memory affinity, since developers do not have to take into consideration the memory management. In this approach, the operating

system is responsible for optimizing all memory allocation and placement. *First-touch* is the default policy in the Linux operating system to manage memory affinity on ccNUMAs. This policy places data on the node that first accesses it [2]. To assure 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 and if threads are not frequently scheduled to different cores/cpus. In case of irregular applications (threads do not always access the same data), *first-touch* will result in a high number of remote accesses.

Currently, there are some proposals concerning new memory policies for Linux. For instance, in [4–6], the authors have designed and have implemented the *on-next-touch* memory policy. This policy allows more local accesses, since each time a thread touches a data, the data migrates when needed. Its performance evaluation has shown good performance gains only for applications that have a single level of parallelism and large amount of data (e.g., matrices which size are higher than 8K x 8K). In case of multiple levels of parallelism (nested parallelism), each thread may create other threads. When these threads share a significant amount of data, several data migration can be performed, since each thread may be in a different machine node. These data migrations have presented an important overhead and they usually have lowered the application performance gains. Moreover, for small amount of data, *on-next-touch* policy have also not presented a good performance since the overhead with migrations is more expensive than the cost of remote accesses.

## 3 Minas

Minas [7] is an efficient and portable framework that allows developers to manage memory affinity in an explicit or automatic way on large scale ccNUMA platforms. In this work, efficiency means fine control of memory accesses for each application variable and similar performance on different ccNUMA platforms. As portability, we mean architecture and compiler abstraction and none or minimal source code modifications.

This framework is composed of three main 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 mechanism. The last module, numarch, is responsible for extracting several information about the target platform. This module can be used by the developer to consult some important information about the architecture and it is also used by Minas-MAi and Minas-MApp mechanisms.

Minas differs from other memory affinity solutions [2, 3] in at least four aspects. First of all, Minas offers code portability. Since numarch provides architecture abstraction, the developer do not have to specify nodes that will be used by Minas to place data. 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).

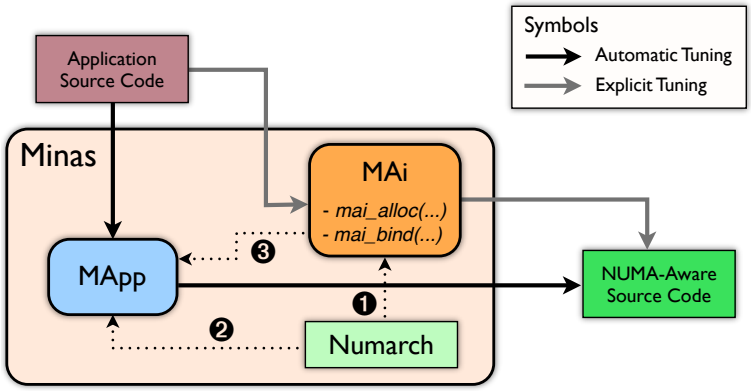


**Fig. 1.** Overview of Minas.

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 (gray arrows) or the automatic one (black arrows). 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.

Depending on the mechanism, numarch is used to retrieve different information. In explicit mechanism, Minas-MAi retrieves from numarch the number of nodes and cpus/cores as well as theirs physical identifiers in order to apply memory policies (dashed arrow 1). Differently, in the automatic mechanism, Minas-MApp gets from numarch the machine's NUMA factor, interconnection bandwidth, cache subsystem information and the amount of free memory of each node. These information are then used by the heuristic function to determine a suitable memory policy (dashed arrow 2). The chosen memory policy will be applied by using Minas-MAi memory policy functions (dashed arrow 3).

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. Minas supports C/C++ and Fortran and the following compilers: Intel, GNU and Portland.

### 3.1 MAi: Memory Affinity interface

MAi (Memory Affinity interface) is an API (Application Programming Interface) that provides a simple way of controling memory affinity [8]. 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. All MAi functions are array-oriented, since MAi was designed for numerical scientific HPC applications.

The most important group of functions on MAi is the memory policies group, since it is responsible for assuring memory affinity. The interface implements eight memory policies that have as their memory affinity unit an array. The memory policies of MAi can be divided in three groups: bind, cyclic and random. Bind memory policies optimize latency, by placing data and threads as close as possible. Both, random and cyclic groups optimize bandwidth of ccNUMA platforms, since they minimize interconnect and memory contention.

Bind group has two memory policies, *bind_block* and *bind_all*. In *bind_block* memory policy, data is divided into blocks depending on the number of threads that will be used and on their placement within the machine. In *bind_all* memory policy, data is placed in one or a set of restrict nodes. Cyclic group is composed by *cyclic*, *skew_mapp* and *prime_mapp* memory policies. In *cyclic*, data is placed according to a linear round-robin distribution, using one memory page per round. In the *skew_mapp* memory policy, a page $i$ is allocated on the node $(i + \lfloor i/M \rfloor + 1)$ *mod* $M$, where $M$ is the number of memory banks. The *prime_mapp* policy uses a two-phase strategy. In the first phase, the policy places data using *cyclic* policy on $(P)$ virtual memory banks, where $P$ is a prime greater or equal to $M$ (real number of memory banks). In the second phase, the memory pages previously placed on virtual memory banks are reordered and placed on real memory banks also using the *cyclic* policy. In *random* policy, memory pages are placed randomly on CC-NUMA nodes, using a random uniform distribution.

The data distribution over the machines nodes can be performed using the entire array or an array tile (blocks distribution). A tile is a sub array which size can be specified by the user or automatically chosen by MAi. Such memory policies allows developers to express different memory access operations, such as write-only, read-only or read/write.

MAi also allows the developer to change the memory policy applied to an array during the application execution, allowing to express different patterns. 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 of an array (specified by the user).

### 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 and platform characteristics at compile time. Its main characteristics are its simplicity of use (automatic NUMA-aware tuning, no manual modifications) and its platform/compiler independence.

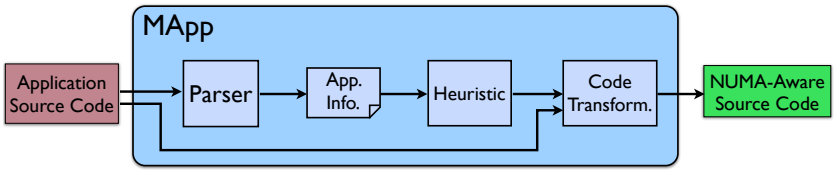


**Fig. 2.** Overview of MApp code transformation process.

The code transformation process is divided into four steps (Figure 2). Firstly, it scans the application source code to obtain information about variables (App Info.). 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, 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 including Minas-MAi specific functions for allocation and data placement.

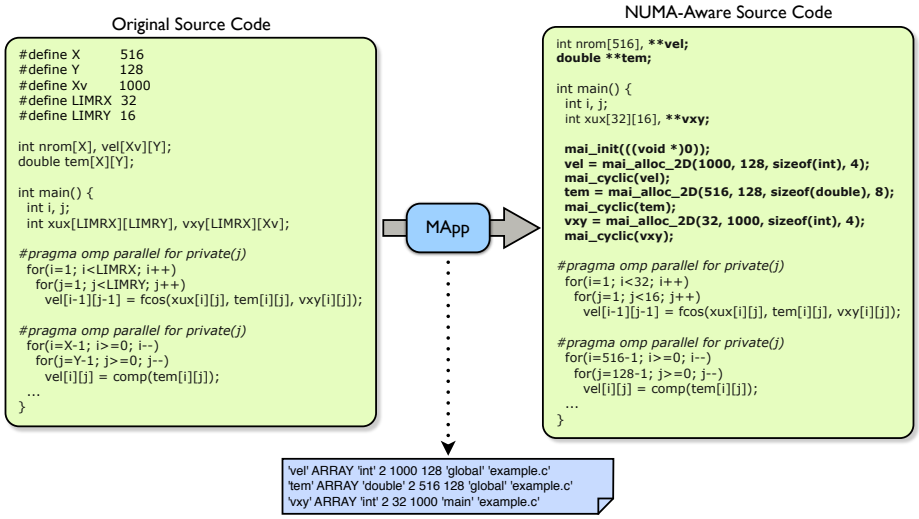


**Fig. 3.** Example of MApp source code transformation.

The most important step of MApp automatic tuning process is the strategy used to decide which memory policy will be applied for each array. Based on empirical data from our previous works and experiments [8–10], 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 and small NUMA factor (ratio between remote latency and local latency to access data), the heuristic will apply cyclic memory policies. On the contrary, on platforms with low number of interconnections and high NUMA factor, the heuristic will opt for *bind_block* memory policies. Figure 3 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 integers *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, we suppose that the target ccNUMA platform has a small NUMA factor (remote latency is low) and a bandwidth problem for interconnection among 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.

### 3.3 Numarch: NUMA Architecture Module

The numarch module has an important role for Minas, since it retrieves the machine information that are necessary to place data on memory banks. This module extracts information about the interconnection network (number of links and bandwidth), memory access costs (NUMA factor and latency) and architecture characteristics (number of nodes, cpus/cores and cache subsystem). To retrieve such information, numarch parses the /sys/devices/ file system of the operating system. The retrieved information is stored in temporary files on the /tmp/ of the operating system. Using such information Minas-MApp places data among the machine nodes reducing latency costs (less remote accesses) and optimizing bandwidth (interconnect contention and memory contention).

This module can also be used as a library, since it provides some high level functions that can be called on the application source code to get some information of the target NUMA machine. The library is composed by a set of functions to retrieve information such as number of nodes, cache size, total of free memory on each node, number of cores per processor, the node of a core/cpu and cores and cpus on a node. Such information can be used by the developer to better understand the machine topology and characteristics.

## 4 Performance Evaluation

In this section, we present the performance evaluation of Minas and compare its results with other three memory affinity solutions for Linux based platforms.

We first describe the two ccNUMA platforms used in our experiments. Then, we describe the two numerical scientific applications (ICTM [10] and Ondes 3D [9]) and their main characteristics. Finally, we present the results and their analysis.

### 4.1 Cache-Coherent NUMA 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). The NUMA factor for this platform varies from **1.2** to **1.5**. The compiler that has been used was the GCC (version 4.3). A schematic representation of this machine is given in Figure 4 (a). We have chosen to use the name **Opteron** for this platform.

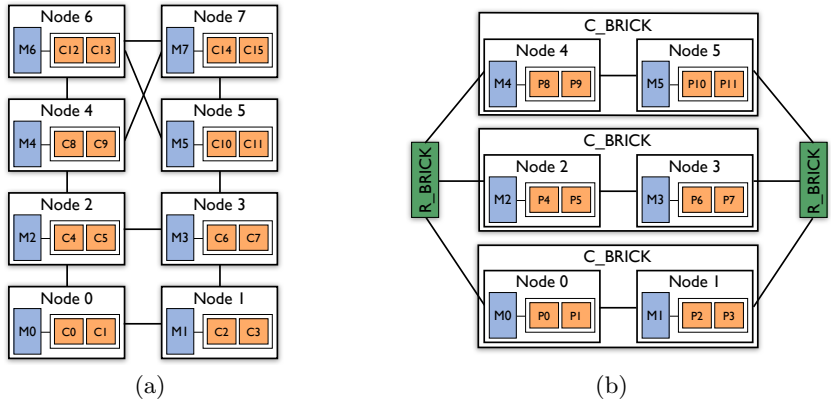


**Fig. 4.** NUMA Platforms: (a) Opteron (b) SGI.

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). The NUMA factor for this platform varies from **1.2** to **1.3**. The compiler that has been used was the ICC (version 9.0). A schematic representation of this machine is given in Figure 4 (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.

### 4.2 Numerical Scientific Parallel Applications

In this section, we present applications Interval Categorizer Tessellation Model (ICTM) [10] and Simulation of Seismic Wave Propagation (Ondes 3D)[9]. Such applications represent important memory-bound numerical scientific problems. The applications have been implemented in C with OpenMP.

**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 [10]. 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 5 (a), the algorithm basically uses nested loops with short and long distance memory accesses (Figure 5 (b)) during the computation phases.

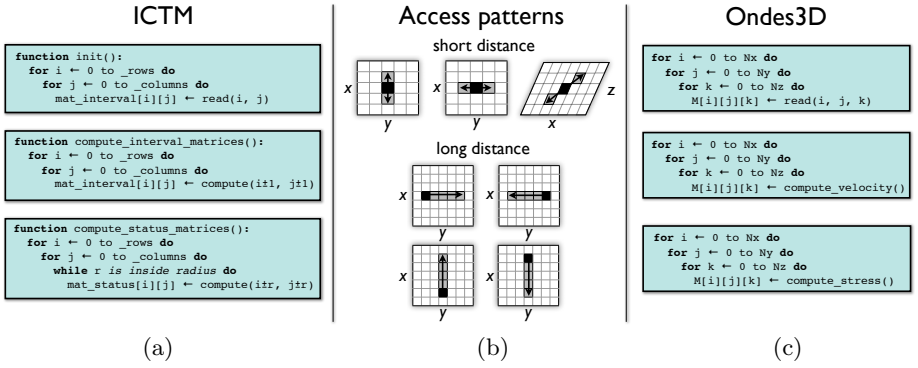


**Fig. 5.** Access patterns: ICTM and Ondes 3D.

**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) [9]. Figure 5 (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 5 (b).

### 4.3 Experimental Results

In this section we present results that have been obtained 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. Our results are organized by application (ICTM and Ondes 3D). For each application, 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*).

Regarding the explicit memory affinity solutions, we have changed applications source codes (Minas-MAi and *libnuma*) or their executions parameters (*numactl*). In order to use Minas-MAi and *libnuma*, the developer must add specific data management functions. The results with Minas-MAi have been obtained by applying the most suited memory policy for each array of the application.

Depending on the application and platform, we have chosen one of the following memory policies (*cyclic*, *prime_mapp* and *bind_block*). The first two memory policies are ideal for irregular applications (ICTM) over ccNUMA platforms that have a small NUMA factor, since they spread data among nodes. The latter memory policy is suitable for regular applications where threads always access the same data set (Ondes 3D). 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 ICTM whereas the *numa_tonode_memory()* function has been used for Ondes 3D. The last explicit solution, *numactl*, does not require source code modifications. 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.

Figure 6 shows the speedups for ICTM on Opteron and SGI platforms with the automatic (Figure 6 (a) and (b)) and the explicit (Figure 6 (c) and (d)) memory affinity solutions. As it can be observed, Minas has outperformed all other memory affinity solutions on both platforms.

Considering the automatic solutions applied to ICTM, Minas-MApp has presented satisfactory results on both platforms (Figure 6 (a) and (b)). Minas-MApp heuristic has chosen *cyclic* memory policy to control data allocation and placement on both platforms. The chosen policy has resulted in better performance gains than *first_touch* (on average, 10% Opteron and 8% on SGI). After a careful

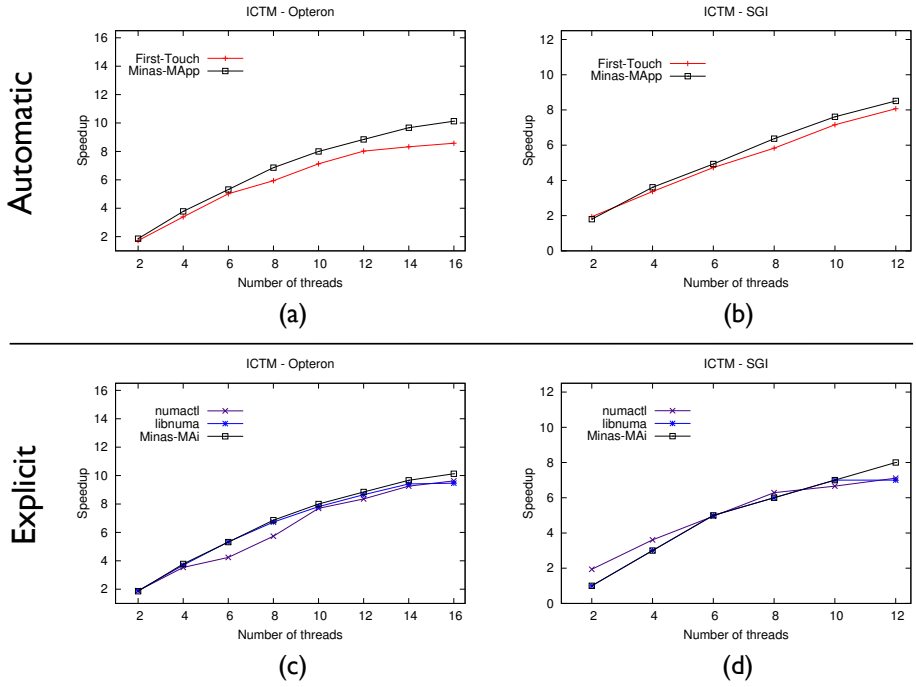


**Fig. 6.** Performance of ICTM on Opteron and SGI platforms.

analysis of these results and application characteristics, we have concluded that *first_touch* policy has generated more remote accesses.

The explicit solutions have presented different behaviors depending on the platform (Figure 6 (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*). It can be explained by the fact that *libnuma* and *numactl* also offer a similar policy, named interleave. It seems that the slight performance gains of Minas-MAi are due to the array optimizations (specialized allocation functions and false sharing reduction). On SGI, Minas-MAi has also presented a better performance thanks to the array optimization included in allocation functions and memory policies. In the case of Minas-MAi, different cyclic memory policies (*cyclic* and *prime_mapp*) have presented equivalent performance gains. The network interconnection characteristics (short distance between memory banks) and the small NUMA factor of the platform can explain this insignificant difference. In this figure, we can also observe that Minas-MAi was the most scalable solution on both platforms in comparison to *libnuma* and *numactl*.

In Figure 7, we show the speedups for Ondes 3D application on Opteron and SGI platforms with the automatic (Figure 7 (a) and (b)) and the explicit

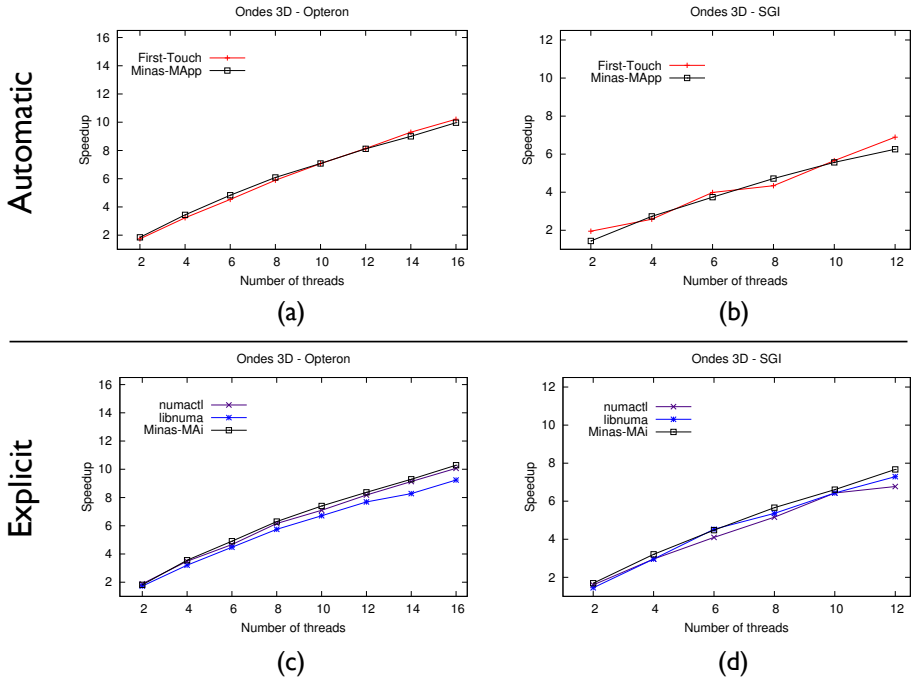


**Fig. 7.** Performance of Ondes 3D on Opteron and SGI platforms.

(Figure 7 (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* and Minas-MApp had similar performance gains. 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 7 (c) and (d)) have shown that *libnuma* and *numactl* have had a worse performance than Minas-MAi. Since this application has a regular memory access, it is important to keep 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 character-

istics. 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.

| | ICTM | Ondes 3D |
|---|---|---|
| Opteron | [0%; 0%] | [0%; 3%] |
| SGI | [0%; 0%] | [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 13%). Considering all the experiments and results, we can conclude that Minas-MApp can be a viable solution when developers do not choose to explicitly modify the application source code.

## 5 Conclusion and Future Work

In this paper we have focused our work on Minas, a memory affinity management framework to deal with memory placement on ccNUMA platforms for numerical scientific HPC applications. We have carried out some experiments over two ccNUMAs to evaluate the efficiency of Minas when used to guarantee memory affinity of two Geophysics applications. Such experiments have shown that Minas has improved the overall performance of applications in comparison with other solutions available on Linux. We have observed that the automatic mechanism of Minas (Minas-MApp) have presented improvements when compared with the Linux native *first_touch* policy. Considering the explicit mechanisms, Minas-MAi has shown better results than other explicit solutions (*numactl* and *libnuma*).

Our future work on Minas includes providing dynamic memory policies, providing a NUMA aware allocator for dynamic data structures [7] as tcmalloc [11], support of memory policies created by developers on Minas-MApp as well as a support for other runtime systems (e.g., Charm++ [12] and TBB [13]) .

## 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. B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on CC-NUMA compute servers," in *ASPLOS-VII: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 279–289.
2. A. Joseph, J. Pete, and R. Alistair, "Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport," in *13th IEEE International Conference on High Performance Computing, Lecture Notes in Computer Science*, 2006, pp. 338–352.
3. A. Kleen, "A NUMA API for Linux," Tech. Rep. Novell-4621437, 2005. [Online]. Available: http://whitepapers.zdnet.co.uk/0,1000000651,260150330p,00. htm
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: 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. C. P. Ribeiro and J.-F. Méhaut, "Minas Project - Memory affInity maNAgement System," 2009. [Online]. Available: http://pousa.christiane.googlepages.com/Minas
8. C. P. Ribeiro, 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*. São Paulo, Brazil: IEEE, 2009.
9. 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*, Lyon, France, 2009.
10. M. Castro, L. G. Fernandes, C. P. Ribeiro, J.-F. Méhaut, and M. 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*, pp. 1–8, 2009.
11. Google, "Google-perftools: Fast, multi-threaded malloc() and nifty performance analysis tools," 2009. [Online]. Available: http://code.google.com/p/google-perftools/
12. A. Gürsoy and L. V. Kale, "Performance and modularity benefits of message-driven execution," *J. Parallel Distrib. Comput.*, vol. 64, no. 4, pp. 461–480, 2004.
13. Intel, "Intel Threading Building Blocks," 2010. [Online]. Available: http://www.threadingbuildingblocks.org/