HAL open science

MAi: Memory Affinity Interface

Christiane Pousa Ribeiro, Jean-François Méhaut

▶ To cite this version:

Christiane Pousa Ribeiro, Jean-François Méhaut. MAi: Memory Affinity Interface. [Technical Report] RT-0359, 2008. inria-00344189v3

HAL Id: inria-00344189
https://inria.hal.science/inria-00344189v3

Submitted on 29 Sep 2009 (v3), last revised 14 Jun 2010 (v6)

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.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

MAI: Memory Affinity Interface

Christiane Pousa Ribeiro — Jean-François Méhaut

N° 0359

December 2008

Thème NUM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--0359--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

MAI: Memory Affinity Interface

Christiane Pousa Ribeiro, Jean-François Méhaut

Thème NUM — Systèmes numériques
Équipe-Projet MESCAL

Rapport de recherche n° 0359 — December 2008 — 22 pages

Abstract: In this document, we describe an interface called MAI. This interface allows developers to manage memory affinity in NUMA architectures. The affinity unit in MAI is an array of the parallel application. A set of memory policies implemented in MAI can be applied to these arrays in a simple way. High-level functions implemented in MAI minimize developers work when managing memory affinity in NUMA machines. MAI's performance was evaluated on two different NUMA machines using three different parallel applications. Results obtained with MAI present important gains when compared with the standard memory affinity solutions.

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

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

MAI: Interface pour la Affinité de Mémoire

Résumé : Ce document décrit MAI, une interface pour contrôler l'affinité de mémoire sur des architecture NUMA. L'unité d'affinité utilisé par MAI est un tableau de l'application. MAI dispose d'un ensemble de politiques de mémoire qui peuvent être appliquées pour le tableau du une application d'une façon simple. Fonctions de haut-niveau mis en œuvre en minimiser les travaux des programmeurs. Les performance de MAI sont évaluée sur deux différents NUMA machines utilisant trois applications parallèles. Les résultats obtenus avec cette interface présente d'importants gains par rapport aux solutions standard.

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

1 Introduction

Parallel applications are generally developed using message passing or shared memory programming models. Both models have their advantages and disadvantages. The message passing model scales better, however, is more complex to use. In the other hand, shared memory programming model is less scalable but easier to be used by the developer. The latter programming model can be used in parallel applications development with libraries, languages, interfaces or compilers directives. Some examples are Pthreads [1], OpenMP [2], TBB [3] etc.

Shared memory programming model is the ideal model for traditional UMA (Uniform Memory Access) architectures, like symmetric multiprocessors. In these architectures the computer has a single memory, which is shared by all processors. This single memory often becomes a bottleneck when many processors accesses it at the same time. The problem gets worse as the number of processors increase, where the single memory controller does not scale.

The limitation of classical symmetric multiprocessors (SMP) parallel systems can be overcome with NUMA (Non Uniform Memory Access) architectures. These architectures have as main characteristics, multiple memory levels that are physically distributed but, seen by the developer as a single memory [4]. NUMA architectures combine the efficiency and scalability of MPP (Massively Parallel Processing) with the programming facility of SMP machines [5]. However, due to the fact that the memory is distributed between the machine nodes, the time spent to access it is conditioned by the distance between the processor and data. The memory access by a given processor could be local (data is close) or remote (it has to use the interconnection network to access the data) [5, 6] .

As these machines are being used as servers for applications that demand a low response time it is important to assure memory affinity on them. Memory affinity is the guarantee that processing units will always have their data close to them. So, to reach the maximum performance on NUMA architectures it is necessary to minimise the number of remote access during the application execution. Which is, processors and data need to be scheduled so as the distance between them are the smallest possible [5, 6, 7, 8].

To reduce this distance and consequently increase the application performance in these architectures, many different solutions were proposed. These solutions are based in algorithms, mechanisms and tools that do memory pages allocation, migration and replication to guarantee memory affinity in NUMA systems for an entire process [5, 9, 6, 7, 10, 11, 12, 4, 13, 14, 15, 4, 16]. However, none of these solutions provides variable/object as memory affinity control unit. In these solutions, memory affinity is assured applying a memory policy for an entire process. Thus, developers can not express the different data access patterns of their application. This can result in a not ideal performance, because memory affinity are related with the program most important variables/objects (importance in means of memory consuption and access).

Thus, to allow different memory policies for each most important variable/object of an application and consequently a better control of memory affinity we developed MAI (Memory Affinity Interface). In this repport we introduce MAI, an interface developed in C that defines some high level functions to deal with memory affinity in NUMA architectures. The main advantages of MAI

are: allow different memory policy for each most important arrays of the application, provide several standard and new memory policies and is a simple way to control memory affinity in parallel applications. In order to evaluate MAI performance we made some experiments with some memory-bound applications (MG from benchmark NAS [17], Ondes 3D [18] and ICTM [19]) in two different NUMA machines. We then compared the results obtained with MAI with Linux standard policy for NUMA machines [11] and the solution most used by developers of parallel application for NUMA machines [17]. This last solution is based in parallel initialization of data, threads and processes touch data to place it close to them (in this work we will call this solution Parallel-Init).

The repport is structured as follows. In section 2 we describe the previous solutions in memory affinity management for NUMA architectures. Section 3 depicts our interface and its main functionalities. In section 4 we present and discuss the performance evaluation of MAI. In the last section we present our conclusions and future works.

2 Memory Affinity Management Solutions

A lot of work has been donne in 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 works.

2.1 Memory Affinity Policies

In the work [9], authors present a new memory policy called *on-next-touch*. This policy allows data migration in the second time of a thread/process touch it. Thus, threads can have their data in the same node, allowing more local access. The performance evaluation of this policy was donne 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.

In [20], 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 [16] present two algorithms to do page migration and assure memory affinity in NUMA machines. These algorithms use information get from kernel scheduler to do migrations. The performance evaluation show gains of 264% considering existed solution as comparison.

2.2 Memory Affinity with OpenMP Directives

In [21], authors present a strategy to assure memory affinity using OpenMP in NUMA machines. The idea is to use information about schedule of threads and 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 in the kernel level (with system call and NUMA APIs) [4].

The user level support allows the programmer to specify a policy for memory placement and threads scheduling for an application. The advantage of using this support is that the programmer does not need to modify the application code. However, the chosen policy will be applied in the entire application and we can not change the policy during the execution.

The API NUMA is an interface that defines a set of system calls to apply memory policies and processes/threads scheduling. In this solution, the programmer must change the application code to apply the policies. The main advantage of this solution is that we can have a better control of memory allocation.

The use of system calls or user level tools to manage memory affinity in NUMA machines can generate some gains. However, developers must know the application and architecture characteristics. To use system calls, developers need to change the application source code and make the memory management by themselves. This is a complex work and depending on the application we can not achieve expressive gains. User level solutions require from developers several shell commands and do not allow fine control over memory affinity.

2.4 Conclusion on Related Works

None of the solutions presented in this section allow developers manage memory affinity in NUMA machines using the most important arrays of the application. Additionally, complex code or modification are necessary to do memory affinity management.

3 MAi

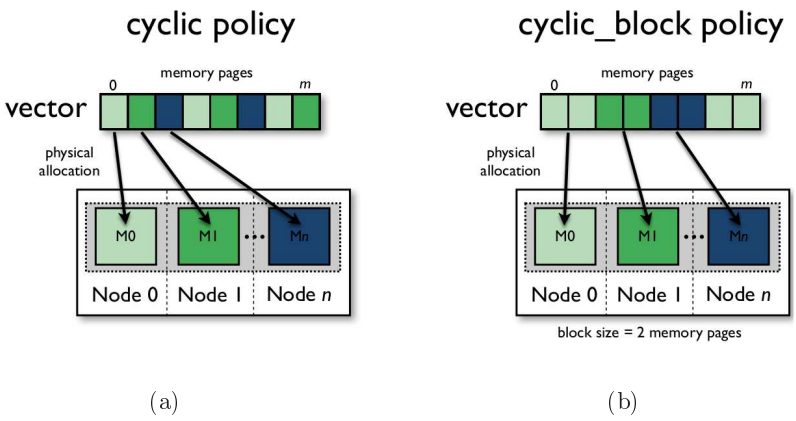
In this section we describe MAI and its features. We first describe the interface proposal and its main contributions. Then, we explain its high level functions

and memory policies. Finally, we describe its implementation details and usage. MAI can be download from [22].

3.1 The Interface

MAI is a library developed in C that defines some high level functions to deal with memory affinity in applications executed on NUMA architectures. This library allows developers to manage memory affinity considering the most important arrays of their applications (importance in terms of memory consumption and access). This characteristic makes memory management easier for developers because they do not need to care about pointers and pages addresses like in the system call APIs for NUMA (libnuma in Linux for example [4]). Furthermore, with MAI it is possible to have a fine control over memory affinity; memory policies can be changed through application code (different policies for different phases). This feature is not allowed in user level tools like numactl in Linux.

MAI main characteristics are: (i) simplicity of use (less complex than other solutions: NUMA API, numactl etc) and fine control of memory (several array based memory policies), (ii) portability (hardware abstraction) and (iii) performance (better performance than other standard solutions).

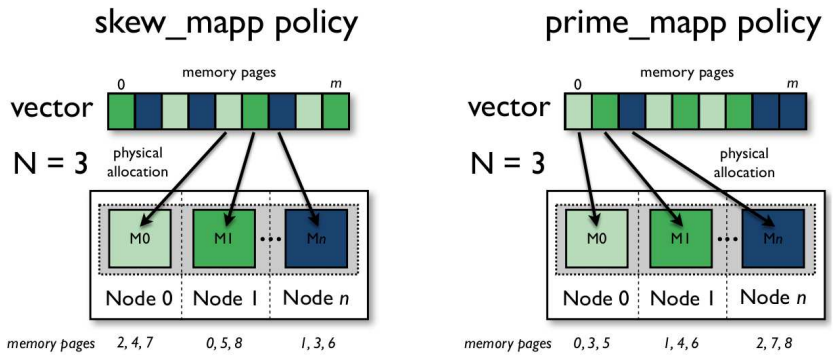


(a) (b)

Figure 1: Cyclic memory policies: (a) cyclic and (b) cyclic_block.

The library has seven memory policies: *cyclic*, *cyclic_block*, *prime_mapp*, *skew_mapp*, *bind_all*, *bind_block* and *random*. In *cyclic* policies the memory pages that contain the array are placed in physical memory making a round with the memory blocks. The main difference between *cyclic* and *cyclic_block*(block can be a set of rows or columns of the array) is the amount of memory pages used to do the cyclic process (Figure 1). In *prime_mapp* memory pages are allocated in virtual memory nodes. The number of virtual memory nodes are chosen by the developer and it has to be a prime number (Figure 2 (b) with prime number equal to 5). This policy tries to spread memory pages through the NUMA machine. In *skew_mapp* the memory pages are placed in NUMA machine in a round-robin way (Figure 2 (a)). However, in every round a shift is done with the memory node number. Both,*prime_mapp* and *skew_mapp*

can be used in blocks, like *cyclic_ block*, where a block can be a set of rows or columns. In *bind_ all* and *bind_ block* (block can be a set of rows or columns of the array) policies the memory pages that contain the array data are placed in memory blocks specified by the application developer. The main difference between *bind_ all* and *bind_ block* is that in the latter, pages are placed in the memory blocks that have the threads/process that will make use of them(Figure 3). The last memory policy is *random*. In this policy, memory pages that contain the array are placed in physical memory in a random uniform distribution. The main objective of this policy is to minimize the memory contention in the NUMA machine.



(a) (b)

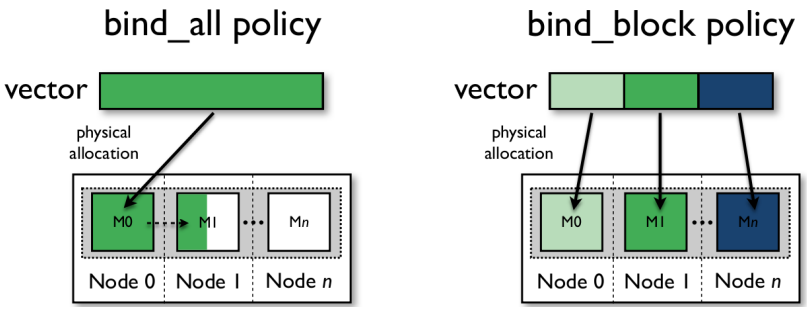
Figure 2: Cyclic memory policies: (a) skew_mapp and (b) prime_mapp.

These memory policies can be combine during the application steps to implement a new memory policy. Arrays generally are accessed in different ways during the application. Thus, it is important give to the developer the opportunity to develop/create a memory policy for his application. The majority of the new proposed memory affinity algorithms do not allows memory policies changes during the applications execution. Also, in MAI it is also possible to do memory migration to optimize any incorrect memory placement.

3.2 High-Level Functions

To develop applications using MAI the programmer just have to use some high-level functions. The functions are divided into five groups: system, allocation, memory, threads and statistic.

System Functions: the system functions are divided in functions to configure the interface and functions to see some information about the platforms. The configuration functions must be called in the begin (mai_init()) and end (mai_final()) of the application code. The function init() is responsible for setting the nodes and threads Ids that will be used for memory/thread policies and for the interface start. This function can receive as parameter a file name. The file is an environment configuration file that gives information about the memory blocks and processors/cores of the underlying architecture. If no file



(a) (b)

Figure 3: Bind memory policies: (a) bind_all and (b) bind_block.

name is give the interface chose which memory blocks and processors/cores to use. The final() function is used to finish the interface and free any data that was allocated by the developer. The other system functions are used to extract information from the target architecture and application.

```
void mai_init(char filename[]);
void mai_final();

int   mai_get_num_nodes();
int   mai_get_num_threads();
int   mai_get_num_cpu();
unsigned long*  mai_get_nodes_id();
unsigned long*  mai_get_cpus_id();
void   mai_show_nodes();
void   mai_show_cpus();
```

Allocation Functions: these functions allows the developer allocate arrays of C primitive types (CHAR, INT, DOUBLE and FLOAT) and not primitive types, like structures. The functions need the number of items and the C type. Their return is a void pointer to the allocated data. The memory is always allocated sequentially, like a linear (1D) array. This strategy allow better performance and also make easer to define a memory policy for an array. Example presented in Figure 4.

```
void* mai_alloc_1D(int nx,size_t size_item,int type);
void* mai_alloc_2D(int nx,int ny,size_t size_item,int type);
void* mai_alloc_3D(int nx,int ny,int nz,size_t size_item,int type);
void* mai_alloc_4D(int nx,int ny,int nz,int nk,size_t size_item, int type);
void   mai_free_array(void *p);
```

Memory Policies Functions: these functions allows the developer to select a memory policy for an application and to migrate data between the nodes. There are seven differents policies: *cyclic*, *cyclic_block*, *prime_mapp*, *skew_mapp*,

```
#include <mai.h>

#define N 1000

int main(int argc, char* argv[])
{
  int i,j;
  double **tab;

  //set information about cpus and nodes
  mai_init(argv[1]);

  tab = (double**)mai_alloc_2D(N,N,sizeof(double),DOUBLE);

  //just some operations
  #pragma omp parallel for
  for(i=0;i<N;i++)
    for(j=0;j<N;j++)
    {
       tab[i][j]= ....;
    }

 //finalize structures and library
 mai_final();
}
```

Figure 4: Alloc functions example

bind_all, *bind_block* and *random*. It is also possible to migrate rows and columns of an array between the nodes of the NUMA system using the functions migrate_*. Example presented in Figure 5.

```
void mai_cyclic(void *p);
void mai_skew_mapp(void *p);
void mai_prime_mapp(void *p,int prime);
void mai_cyclic_rows(void *p,int nr);
void mai_skew_mapp_rows(void *p,int nr);
void mai_prime_mapp_rows(void *p,int prime,int nr);
void mai_cyclic_columns(void *p,int nc);
void mai_skew_mapp_columns(void *p,int nc);
void mai_prime_mapp_columns(void *p,int prime,int nc);

/*Bind memory policy - function prototypes*/
void mai_bind_all(void *p);
void mai_bind_rows(void *p);
void mai_bind_columns(void *p);

/*Random memory policy - function prototypes*/
void mai_random(void *p);
void mai_random_rows(void *p, int nr);
void mai_random_columns(void *p, int nc);
```

```
/*Migration - function prototypes*/
void mai_migrate_rows(void *p,unsigned long node,int nr, int start);
void mai_migrate_columns(void *p,unsigned long node,int nc, int start);
void mai_migrate_scatter(void *p,unsigned long nodes);
void mai_migrate_gather(void *p,unsigned long node);
```

```
#include <mai.h>

#define N 1000

struct mydata{
int i;
char *a;
};

struct mydata *t;

int main(int argc, char* argv[])
{
  int i,j,k;

  //set information about cpus and nodes
  mai_init(argv[1]);

  //allocation data
  t = (struct mydata*) mai_alloc_1D(N,sizeof(struct mydata),

  mai_bind_columns(t);

 //just some operations
#pragma omp parallel for
  for(i=0;i<N;i++)
    {
       t[i].i= .....;
    }

 //finalize structures and library
 mai_final();
}
```

Figure 5: Memory policies functions example

Thread Policies Functions: these functions allows the developer to bind or migrate a thread to a cpu/core in the NUMA system. The function bind_threads use the threads identifiers and the cpu/core mask (giving in the configuration file) to bind threads.

```
void bind_threads();
void bind_thread_posix(int id);
void bind_thread_omp(int id);
void mai\_migrate_thread(pid_t id,unsigned long cpu);
```

Statistics Functions: these functions allows the developer to get some information about the application execution in the NUMA system. There are

statistics of memory and threads placement/migration and overhead from migrations.

```
/*Page information - function prototype*/
void mai_print_pagenodes(unsigned long *pageaddrs,int size);
int mai_number_page_migration(unsigned long *pageaddrs,int size);
double mai_get_time_pmigration();

/*Thread information - function prototype*/
void mai_print_threadcpus();
int mai_number_thread_migration(unsigned int *threads,int size);
double mai_get_time_tmigration();
```

3.3 Implementation Details

MAI interface is implemented in C for Linux based NUMA systems. Developers who wants to use MAI have to program their applications using shared memory programming model. The allowed shared memory libraries are: Posix threads or OpenMP for C/C++ languages. To use this interface, *libnuma* must be installed in the NUMA system. Figure 6 present the architecture of the interface.

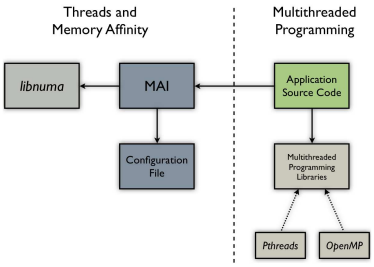


Figure 6: MAI Architecture

To guarantee memory affinity, all memory management is donne using MAI structures. These structures are loaded in the initialization of the interface (with the init() function) and are used in the rest of application execution. All allocations and memory policies algorithms consider the NUMA architecture before do any allocation and placement of memory pages in the memory blocks. A memory policy set for a variable/object of the application can be changed during the application steps. To do this developers just have to call the memory policy function for the variable/object and MAI will change it.

Threads scheduling in MAI is donne using one thread per processor/core strategy. This minimizes memory contention problem that exists in some NUMA architectures. If the number of threads are bigger then the number of processors/cores then, more threads per processor/core will be used.

4 Performance Evaluation

In this section we present the performance evaluation of MAI. We first describe the two NUMA platforms used in our experiments. Then, we describe the applications and their main characteristics. Finally, we present the results and their analysis.

4.1 NUMA Platforms

In our experiments we used two different NUMA architectures. The first NUMA architecture is a sixteen Itanium 2 with 1.6 GHz of frequency and 9 MB of L3 cache memory for each processor. The machine is organized in four nodes of four processors and has in total 64 GB of main memory. This memory is divided in four blocks in each node (16 GB of local memory). The nodes are connected using a FAME Scalability Switch (FSS), which is a backplane developed by Bull [23]. This connection gives different memory latencies for remote access by nodes (NUMA factor from 2 to 2.5). A schematic figure of this machine is given in Figure 7. We will use the name Itanium 2 for this architecture.

The operating system that has been used in this machine was the Linux version 2.6.18-B64k.1.21 and the distribution is Red Hat with support for NUMA architecture (system calls and user API numactl). The compiler that has been used for the OpenMP code compilation was the ICC (Intel C Compiler).

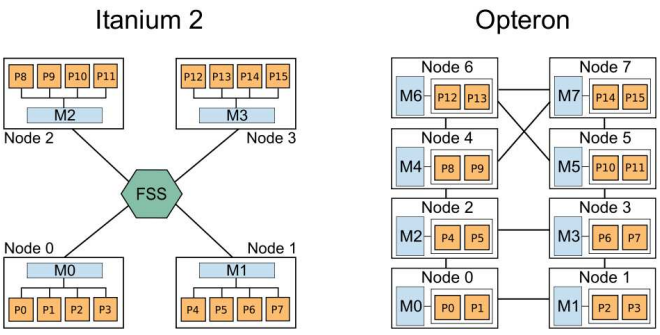


Figure 7: NUMA platforms.

The second NUMA architecture used is an eight dual core AMD Opteron with 2.2 GHz of frequency and 2 MB of cache memory for each processor. The machine is organized in eight nodes and has in total 32 GB of main memory. This memory is divided in eight nodes (4 GB of local memory) and the system page size is 4 KB. Each node has three connections which are used to link with other nodes or with input/output controllers (node 0 and node 1). These connections give different memory latencies for remote access by nodes (NUMA factor from 1.2 to 1.5). A schematic figure of this machine is given in Figure 7. We will use the name Opteron for this architecture.

The operating system that has been used in this machine is Linux version 2.6.23-1-amd64 and the distribution is Debian with support for NUMA archi-

tecture (system calls and user API numactl). The compiler that has been used for the OpenMP code compilation was the GNU Compiler Collection (GCC).

4.2 Applications

In our experiments we used three different applications: one from benchmark NAS (Conjugate Gradiente [17]), Ondes 3D (seismology application [24]) and ICTM (Interval Categorization Tessellation model [25]). All the applications have as main characteristics high memory consumption, number of data access and regular and/or irregular data pattern access. In this work, we consider as regular pattern access, when threads of an application access always the same data set. On the contrary, on the irregular data pattern access, threads of an application do not always access the same data set. Threads or process of an application do not access the same data set during the initializations and computational steps of the application. All applications were developed in C using OpenMP to code parallelization.

One of the three applications is a kernel from NAS benchmark [17] called MG. This kernel were selected because they are representative in terms of computations and present memory important characteristics. MG is a kernel that uses a V cycle MultiGrid method to calculate the solution of the 3D scalar Poisson equation. The main characteristic of this kernel is that it tests both short and long distance data movement. Figure 8 present a schema of the application.

```
#pragma omp parallel for
    for (i=0 to i<rows)
     for (j=0 to j<columns)
       for (k=0 to k<planes)
          _mat[i][j][k] = ...

     for (i=0 to i<rows)
    #pragma omp parallel for
     for (j=0 to j<columns)
       for (k=0 to k<planes)
        _mat[i][j][k] = ...


  #pragma omp parallel for
    for (i=0 to i<rows)
     for (j=0 to j<columns)
       for (k=0 to k<planes)
       _mat[i][j][k] = mat[-+i][-+*j][-+*k]
            ...
```

Figure 8: Multi Grid

Ondes 3D is a seismology application simulates the propagation of a seismic in a 3 dimension region [18]. This application has three main steps: data allocation, data initialization and propagation calculus (operations). In the allocation step all arrays are allocated. The initialization is important because the data are touched and physically allocated in the memory blocks. The last step is composed by two big calculus loop. In these loops all arrays are accessed for write an read but, with the same access patterns of the initialization step. Figure 9 present a schema of the application.

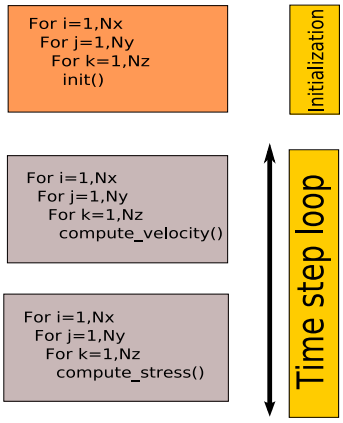


Figure 9: Ondes 3D Application

ICTM is a multi-layered and multi-dimensional tessellation model for the categorization of geographic regions considering several different characteristics (relief, vegetation, climate, land use, etc.) [19]. This application has two main steps: data allocation-initialization and classification computation. In the allocation-initialization step all arrays are allocated and initialized. The second step is composed by four big calculus loops. In these loops all arrays are accessed for write an read but, with the regular and irregular access patterns.Figure 10 present a schema of the application.

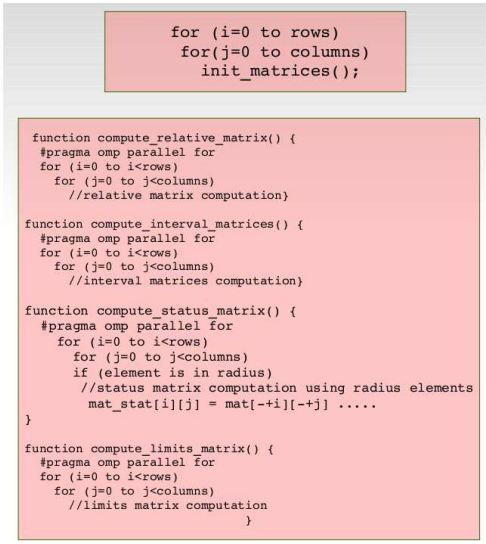


Figure 10: ICTM Application

4.3 Results

In this section we present the results obtained with each application in the two NUMA machines with MAI and the two other solutions (Linux native solution, first-touch and Parallel-Init, first-touch used in a parallel data initialization). The experiments were donne using 2,4,8 and 16 threads and a problem size of 256^2 elements fo MG, 2.0 Gbytes for ICTM and 2.4 Gbystes for Ondes 3D. As performance metrics we selected speedup (Sequential execution time/parallel execution time) and efficiency (speedup/number of threads).

In figure 11(a) we present the speedups and efficiencies for the MG application when executed in Opteron machine. As we can observe for this application and considering an Opteron NUMA machine the better memory policy is *cyclic*. This NUMA machine has as important characteristic a not so good network bandwidth. The optimization of bandwitdh in this machine is more important than the optimization of latency. As the NUMA factor in this machine is small the latency influence in this type of application is not high (irregular pattern access). Thus, distribute the memory pages in a cyclic way avoid contention and consequently the influence of network bandwidth in the application execution. The others memory policies had worst results because they try to optimize latency and as the problem in this machine is contention in network these policies do not present good results.

In figure 11(b) we present the speedups and efficiencies for the MG application for Itanium machine. The memory policy *bind_ block* is the best one, as this machine has a high NUMA factor try to allocate data close to threads that use it is better. The cyclic memory policy had worst results because it try to optimize bandwidth and as the machine has a high NUMA factor this policy (optimize bandwidth)do not present good results. The other two policies, First-Touch and Parallel-Init also do not have good gains. First-touch policy centralizes data on a node so, threads made a lot of remote and concurrent access on the same memory block. On the other hand, Parallel-Init try to distribute data between the machine nodes but, as the access patterns are irregular this strategy do not give good speedups and efficiencies.

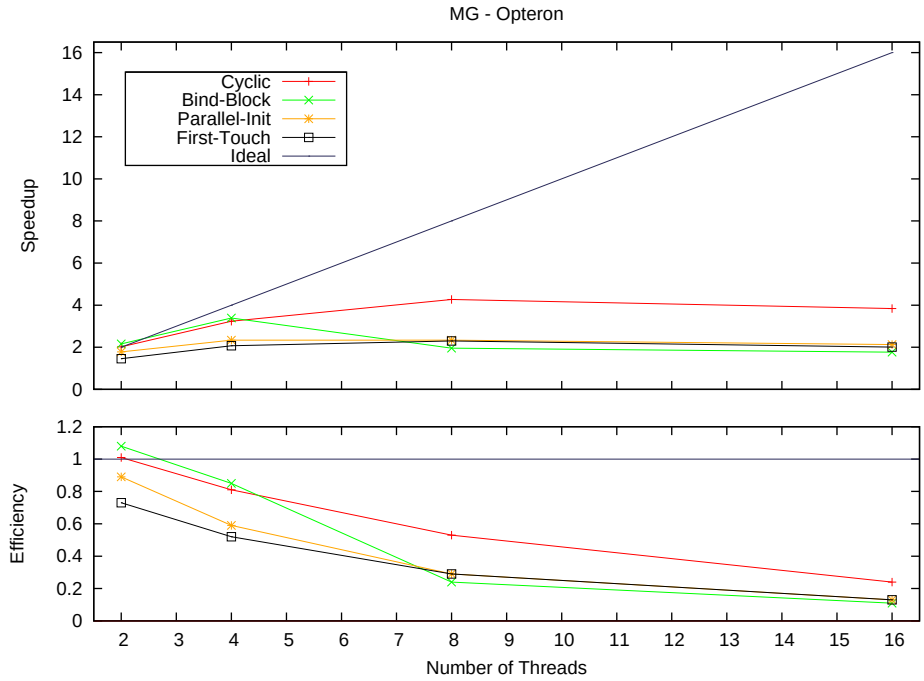
In figures 12(a) and 12(b) we present the results for ICTM application considering the Opteron and the Itanium machines. For this application, as for MG, the better memory policies are *cyclic* for Opteron and *bind_ block* for Itanium.

Opteron has a network bandwidth problem so, it is better to spread the data among NUMA nodes. By using this strategy, we reduce the number of simultaneous accesses on the same memory block. As a consequence of that, we can have a better performance. The *bind_ all* and *bind_ block* policies have presented worse speed-ups for Opteron machine. These policies allows a lot of remote access by threads and also do not consider network bandwidth in memory allocation.

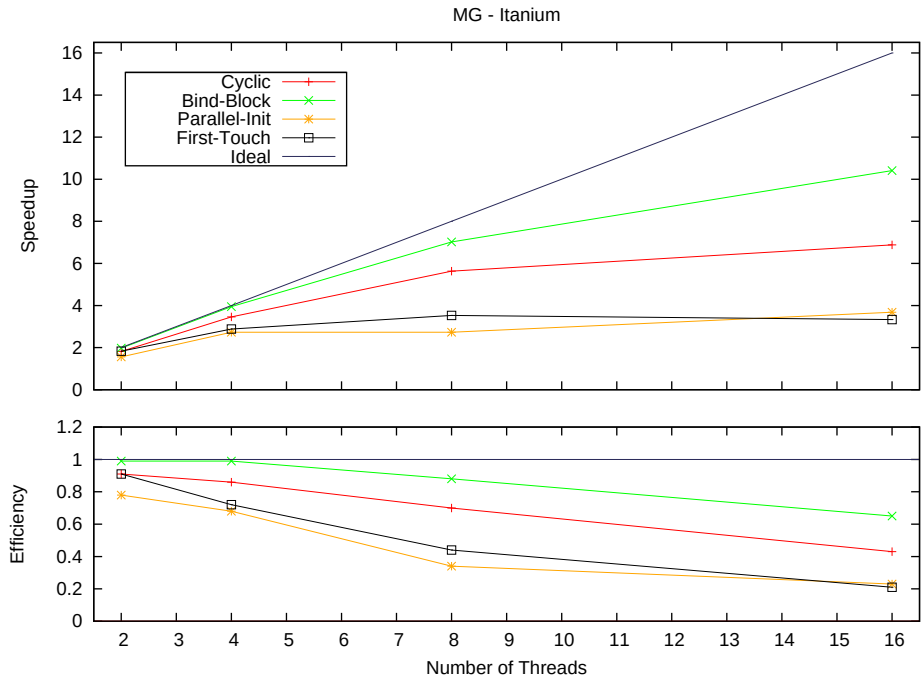
We can see that the results for Itanium are quite different when compared to those obtained over Opteron (Figure 12(b)). By allocating data closer to the threads which computes them, we decrease the impact of the high NUMA factor. As a result, *bind_ block* was the best policy for this machine.

In figures 13(a) and 13(b) we present the results for Ondes 3D application considering the Opteron and the Itanium machines. As we can observe, the results obtained with MAI policies, First-Touch and Parallel-Init solutions were similar. The main reason for this is the regularity of data access patterns of this

application. Threads always access the same set of memory pages. However, there is a little performance gain in the use of *bind_block* policy. This policy place threads and data in the same node, so it avoids remote access. In the other solutions threads can migrate or data are placed in a remote node allowing more remote access.

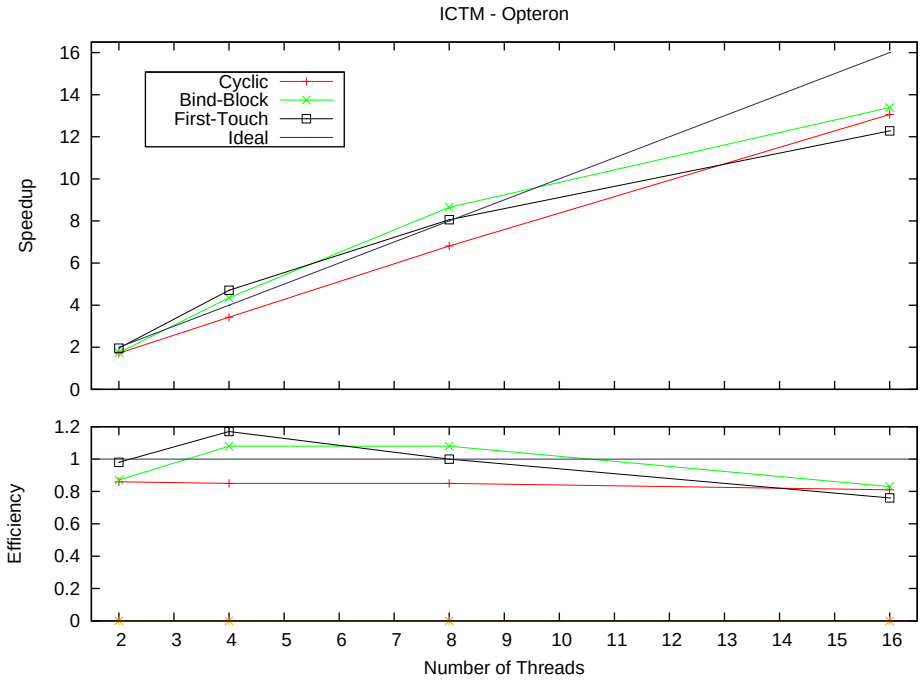


(a) Opteron Speedup-Efficiency

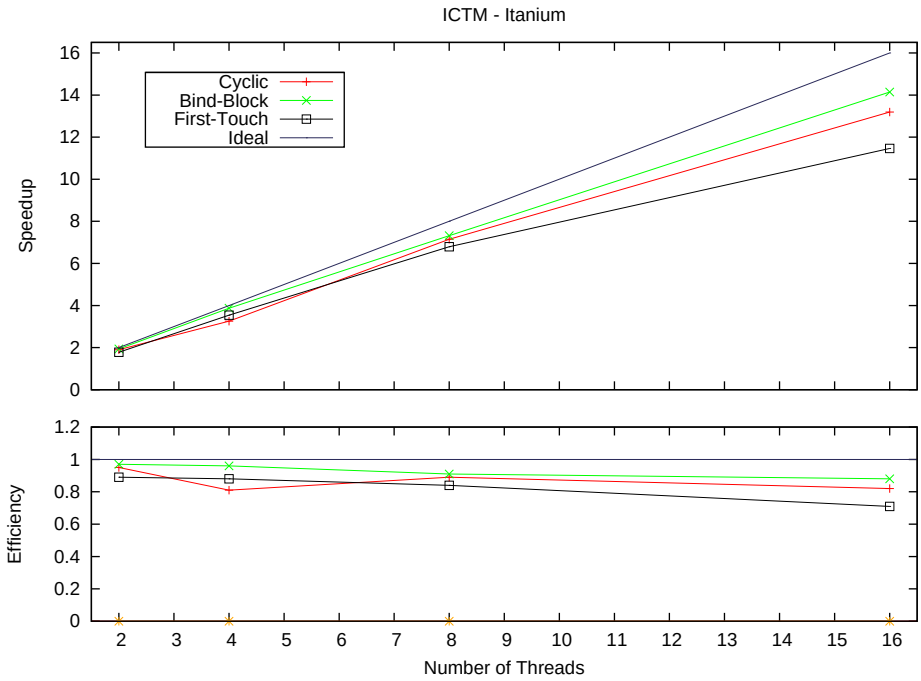


(b) Itanium Speedup-Efficiency

Figure 11: MG Application

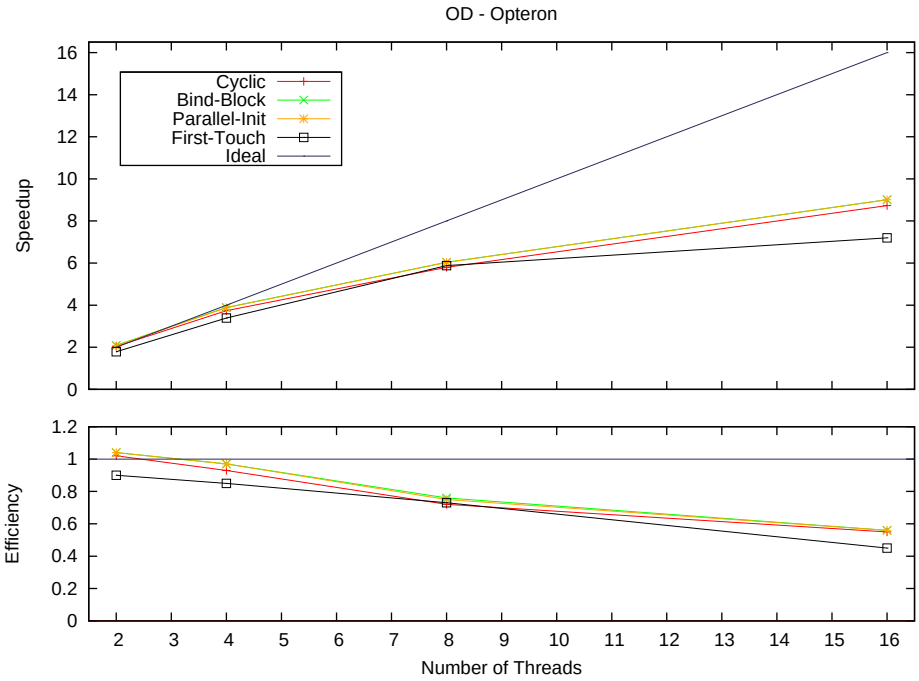


(a) Opteron Speedup-Efficiency

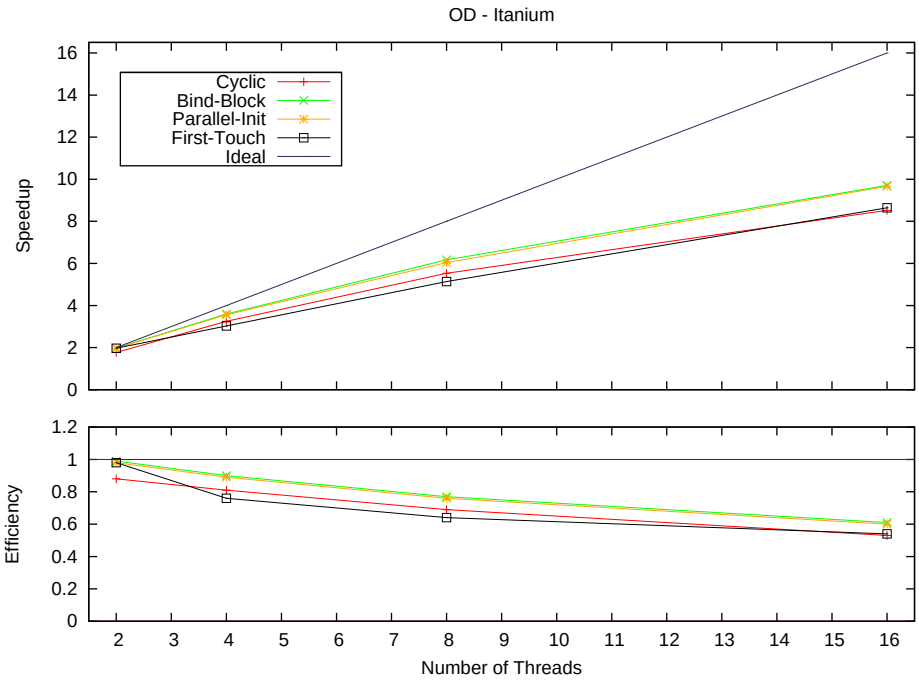


(b) Itanium Speedup-Efficiency

Figure 12: ICTM Application



(a) Opteron Speedup-Efficiency



(b) Itanium Speedup-Efficiency

Figure 13: Ondes 3D Application

5 Conclusion

In this report we presented an interface to manage memory affinity in NUMA machines called MAI. This interface has some memory policies that can be applied to array of an application to assure memory affinity. MAI is also an easy way of manage memory in NUMA machines, high level functions allow developers to change their application with no big work.

The experiments made with MAI show that the interface has better performance than some proposed solutions. We evaluate MAI using three different applications and two NUMA machines. The results show gains of 2%-60% in relation of Linux memory policy and optimized application code version.

The work show that memory has to be manage in NUMA machines and that this management must be easy for the developer. Thus, an interface that helps developers to do this is essencial. As future works we highlight: MAI version that allows developers to implement new memory policies as plug-ins, new performance evaluation of MAI with others applications and integrate MAI in GOMP, the GCC implementation of OpenMP.

References

[1] D. R. Butenhof, "Programming with posix threads," 1997.

[2] OpenMP, "The openmp specification for parallel programming - http://www.openmp.org," 2008. [Online]. Available: http://www.openmp.org

[3] "Threading building blocks-http://www.threadingbuildingblocks.org/," 2008. [Online]. Available: http://www.threadingbuildingblocks.org/

[4] 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

[5] 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.

[6] 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

[7] F. Bellosa and M. Steckermeier, "The Performance Implications of Locality Information Usage in Shared-Memory Multiprocessors," *J. Parallel Distrib. Comput.*, vol. 37, no. 1, pp. 113–121, August 1996. [Online]. Available: http://portal.acm.org/citation.cfm?id=241170.241180

[8] 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.

[9] 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

[10] 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

[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] L. T. Schermerhorn, "Automatic Page Migration for Linux," in *Proceedings of the Linux Symposium*, Linux, Ed., Sydney, Australia, 2007.

[13] 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*. New York, NY, USA: ACM, 1996, pp. 279–289. [Online]. Available: http://portal.acm.org/citation.cfm?id=237090.237205

[14] R. Vaswani and J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors," *SIGOPS Oper. Syst. Rev.*, vol. 25, no. 5, pp. 26–40, 1991.

[15] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "Scheduling and Page Migration for Multiprocessor Compute Servers," *SIGPLAN Not.*, vol. 29, no. 11, pp. 12–24, 1994.

[16] 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.

[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] R. K. S. Silva, C. A. F. D. Rose, M. S. de Aguiar, G. P. Dimuro, and A. C. R. Costa, "Hpc-ictm: a parallel model for geographic categorization," in *JVA '06: Proceedings of the IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 143–148.

[20] 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.

[21] 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.

[22] C. P. Ribeiro and J.-F. Méhaut, "Memory Affinity Interface," INRIA, Tech. Rep. RT-0359, December 2008. [Online]. Available: http://hal.inria.fr/inria-00344189/en/

[23] BULL, "Bull - architect of an open world," 2008. [Online]. Available: http://www.bull.com/

[24] 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.

[25] 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.

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