MAi: Memory Affinity Interface

Christiane Pousa Ribeiro, Jean-François Méhaut

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 — 29 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

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]. On such machines, shared memory is physically distributed in memory banks among the machine nodes. These memory banks are interconnect by a high performance network (hypertransport from AMD, QuickPath from Intel and NUMAlink from SGI). Because of this, time spent to access data is conditioned by the distance between the processor and memory bank where data was placed. The memory access by a given processor can be local (data is close) or remote (it has to use the interconnection network to access the data) [5, 6].

The concept of NUMA machines is not new and it was first proposed on 80's/90's to overcome scalability problems on multiprocessors. Nowadays, this NUMA concept is coming back to assure good memory performance. On new ccNUMAs machines, the number of processing units accessing memory banks is much larger than the older ccNUMA machines (DASH, SGI, KSR ect). These accesses produce more stress on the memory banks, generating more load-balancing issues, memory contention and remote accesses. As these machines are extensively used in HPC, it is important to minimize these memory access costs and then, to guarantee good performance of applications by assuring memory affinity.

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 portability and variable as memory affinity control. 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 variables of scientific numerical high performance application and consequently a better control of memory affinity we propose 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 on ccNUMA (cache-coherent NUMA) architectures. The main advantages of MAi are: it allows different memory policy for arrays of the application, it has several standard and new memory policies, it provides architecture abstraction and it 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]) on two different ccNUMA machines. We then compared the results obtained with MAi with Linux standard policy for NUMA machines [11] and the solution most used by developers, first-touch [17]. This last solution is based in parallel initialization of data, threads 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

Several works have been done since 90's to guarantee memory affinity on NUMA machines. These works have lead to some solutions that can be classified into six groups: compilers affinity support, extensions to languages (OpenMP, HPF), libraries of memory policies (libnuma), memory policies, runtimes (libgomp on linux) and tools (dplace, numactl). In this section we present these groups of related works.

2.1 Compiler Affinity Support

The advent of multi-core NUMA machines have brought new challenges to compilers that support OpenMP codes. To deal with such challenges, compilers have received special abilities such as affinity support. Several compilers have now the capability of distribute threads over the NUMA machine nodes, and consequently also assure memory affinity. Examples of such compilers are Intel C Compiler (ICC), GNU C Compiler (GCC) and Portland Group Compiler (PGI).

By using this support, developers can bind OpenMP threads to processing units in different ways. Binding is generally done before the OpenMP application by using environment variables or interfaces (few modifications on the source code are needed) that allow programmers to specify policies to distribute threads over the physical processing units. However, these policies are related to the machine topology, application, and operating system, and developers must be aware of these characteristics when choosing the affinity policy for the OpenMP application [20, 21].

2.2 Memory Affinity on Parallel Languages

Considering OpenMP, some extensions have been proposed to better place data on the NUMA machines. However, none of these extensions were included on the OpenMP standard because the standard is responsible for defining just how the work can be done in parallel. Additionally, such extensions are supported on a restricted set of compilers and are applied in a static fashion (do not consider data migration at runtime).

In [22], 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.

HPF (High Performance Fortran) has proposed several directives to better distribute data over the platform. In order to maximize page locality, data distribution is performed considering the parallel computations [23, 24]. Contrary to our proposal, on HPF, directives must be explicit included in the application source code by the developer. This can be a difficult task, since the best data distribution is generally related to hardware characteristics. Thus, developers must have a priori knowledge of the platform to better choose the HPF directive.

2.3 Memory Affinity Libraries

The API NUMA is a Library that defines a set of system calls on Kernel and high level functions to apply memory policies and processes/threads scheduling. 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 these solutions, the programmer must change the application code to apply the policies. This is a complex work and depending on the application we can not achieve expressive gains. The main advantage of this solution is that we can have a better control of memory allocation [4].

2.4 Memory Affinity Policies

First-touch is the default memory policy in Linux operating system to manage memory affinity on ccNUMAs. This policy places data on the node that first accesses it [10]. 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. In case of irregular applications (threads do not always access the same data), *first-touch* will result in a high number of remote accesses.

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 [25], 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.

2.5 Memory Affinity aware Runtimes

Dynamic task and data placement for OpenMP applications have been proposed in [26, 27]. In these works, researchers have proposed an extension of GNU OpenMP library, named ForestGOMP. This proposal extracts information about threads and data relationship at runtime to better distribute them over the NUMA machine.

2.6 Memory Affinity Tools

NUMA support is now present in several operating systems, such as Linux and Solaris. This support can be found in the user level as administration tools or shell commands [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. User level solutions require from developers several shell commands and do not allow fine control over memory affinity.

2.7 Conclusion on Related Works

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 are limited to specific NUMA platforms, they do not address different memory accesses (limited set of memory policies) and they do not include optimizations for numerical scientific data (i.e., array data structures).

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 memory policies and its high level system functions. After that, we present some examples of how developers can use MAi to design/create their own memory policy. Finally, we describe its implementation details and usage. MAi can be download from [28].

3.1 The Interface

MAi is an interface that defines some high level functions to deal with memory affinity on scientific numerical applications executed on ccNUMA architectures [29]. This interface allows developers to manage memory affinity considering the arrays of their applications. Such characteristic makes memory management easier for developers because they do not have 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). MAi also allows developers to create memory their own memory policies using a set of high level functions.

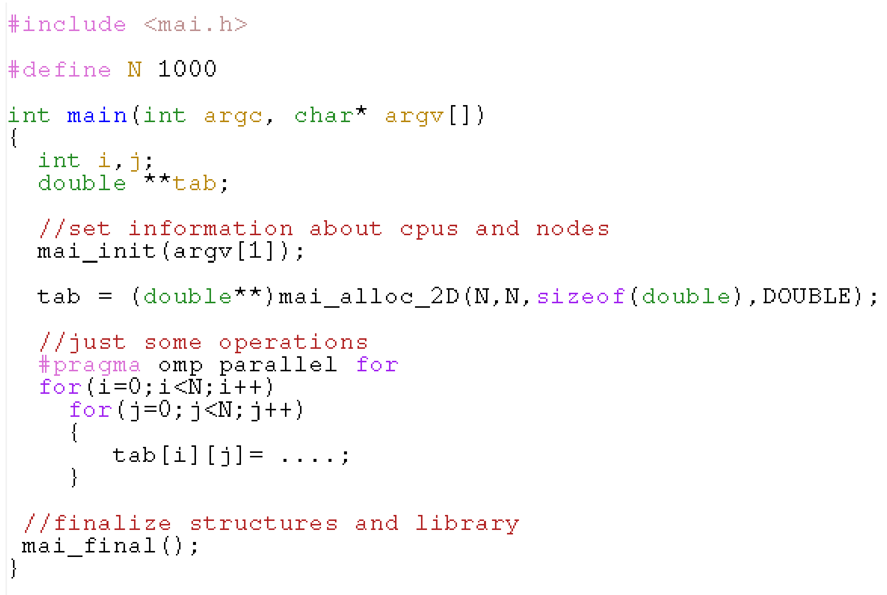
MAi main characteristics are: (i) simplicity of use (less complex than other solutions: NUMA API, numactl etc) and fine control of memory (several array based memory policies), (ii) portability (hardware abstraction) and (iii) performance (better performance than other standard solutions).

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 banks. System functions allows developers to develop/create their own memory policy and to collect system information such as memory banks used by the memory policies, cpus/cores used during the application execution, memory banks and statistics about page migration. Furthermore, MAi has a thread placement mechanism that are used with some memory policies to better assure memory affinity.

3.2 Allocation Functions:

The allocation 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 1.

```
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);
```



```
#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 1: Alloc functions example

3.3 Memory Policies

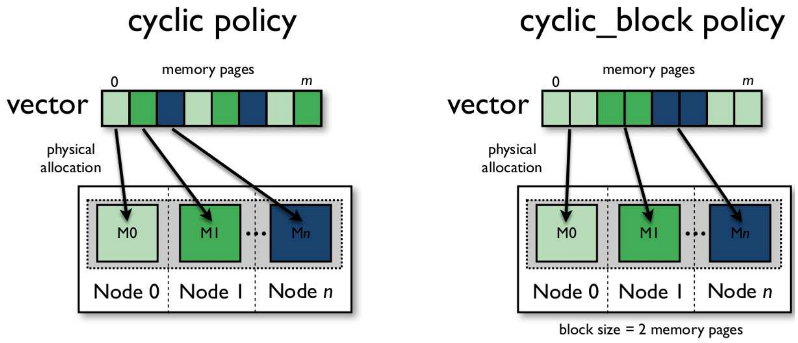
The interface has seven memory policies: *cyclic*, *cyclic_block*, *prime_mapp*, *skew_mapp*, *bind_all*, *bind_block*, *bind_page*, *random* and *next_touch*.

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 2). 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 3 (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 3 (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*, *bind_ block* (block can be a set of rows or columns of the array) and *bind_ page* policies the memory pages that contain the array data are placed in a strict set of memory blocks. The memory blocks can be specified by the application developer or automatic selected by MAi. The main difference between *bind_ all* and *bind_ block*/*bind_ page* is that in the latters, pages are placed in the memory blocks that have the threads/process that will make use of them (Figure 4).

In *random* 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.

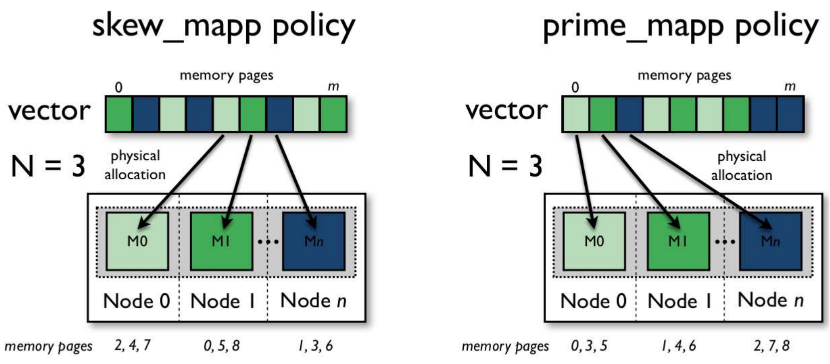
For irregular applications, it is necessary to provide dynamic memory policies that support the irregularity of the application over its phases. *Next_ touch* is a memory policy that allows developers to express such irregularity. On this memory policy, every time that the array is touched by a different NUMA node, its memory pages are migrated to assure data locality.



(a) (b)

Figure 2: Cyclic memory policies: (a) cyclic and (b) cyclic_block.

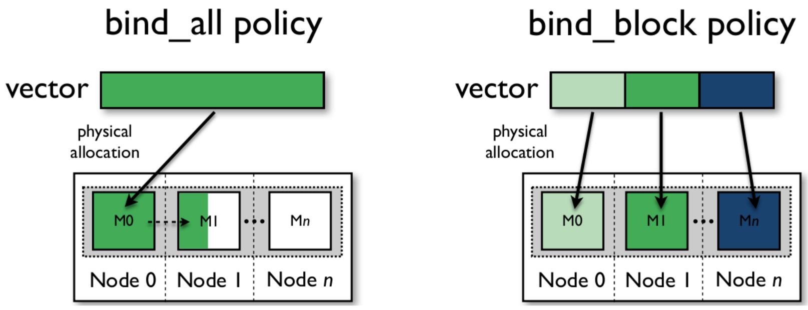
One of the most important features of MAi is that it allows the developer to change the memory policy applied to an array during the application execution. This characteristic allows developers to express different patterns during the application execution. Additionally, MAi memory policies can be combined 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



(a) (b)

Figure 3: Cyclic memory policies: (a) skew_mapp and (b) prime_mapp.

pages (automatically defined by MAi) or a set of rows/columns (specified by the user).



(a) (b)

Figure 4: Bind memory policies: (a) bind_all and (b) bind_block.

Bellow, we present a sniped of memory policies functions declaration. As we can observe, developers just have to provide the array and the block size (if they want to express any pattern access). MAi memory policies will perform data placement considering the available memory banks on the machine and machine characteristics.

```
Interleave memory policies:
void mai_cyclic(void *p);
void mai_skew_mapp(void *p);
void mai_prime_mapp(void *p,int prime);
void mai_cyclic_block(void *p,int block);
void mai_skew_mapp_block(void *p,int block);
void mai_prime_mapp_block(void *p,int prime,int block);
```

```
Bind memory policies:
void mai_bind_all(void *p);
void mai_bind_block(void *p);
void mai_bind_page(void *p);
void mai_ibind_block(void *p, int block);

Random memory policies:
void mai_random_all(void *p);
void mai_random_block(void *p);
void mai_random_block(void *p, int block);

Dynamic memory policies:
void next_touch(void *p);

Memory Migration:
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);

Block = rows, colunms or plans
```

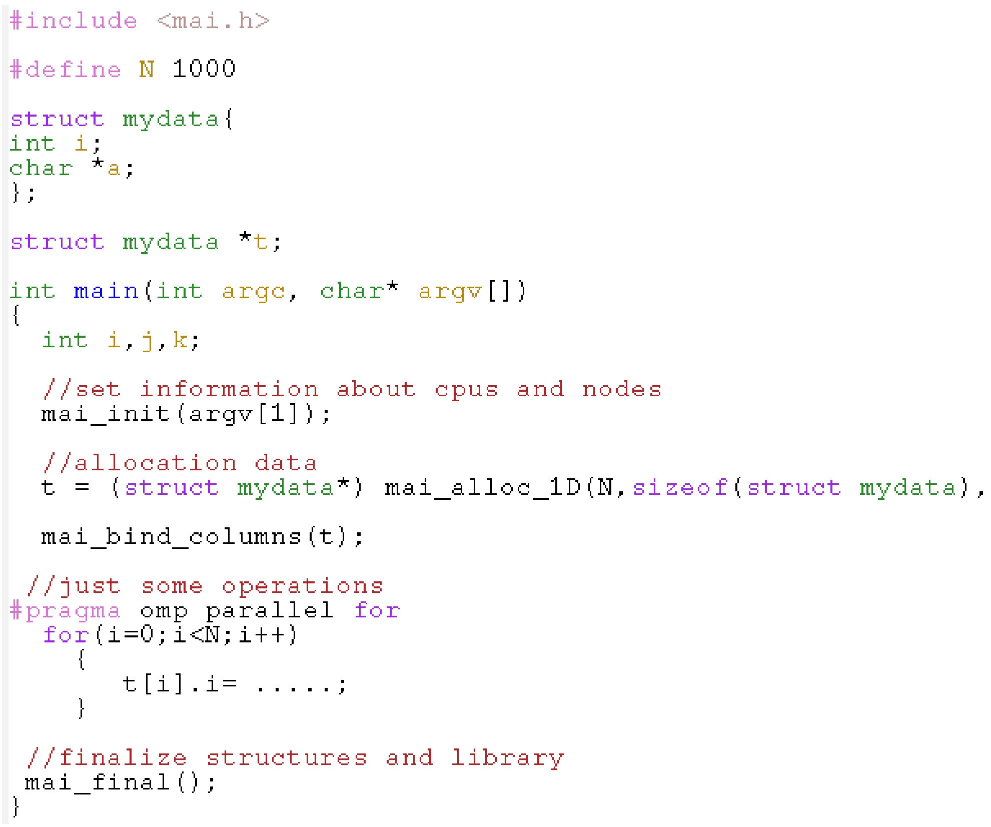
The Figure 5 presents a simple example of an OpenMP application. In this example, some MAi functions were include in the application source code to apply a memory policy to an array. It is important to observe that the array must be allocated using MAi allocators (in this example mai_alloc_1D.

3.4 System Functions

To develop applications using MAi, developers just have to include some high-level functions on their application source code, like allocation and memory policies functions discussed on previous sections. However, different applications have different needs (i.e., memory access types). Additionally, memory subsystem of ccNUMA platforms may be also different and has to be considered when developing a memory policy. Thus, it is necessary to have different memory policies that can adapt to each case.

MAi provides an extended set of system functions that can be used by developers to design/develop their own memory policy. By designing their own memory policy, developers can try different fashions of distribute data among memory banks. And, after that, they can chose the best memory policy and apply it to their application. The system functions allows developers to retrieve information of the architecture and to distribute data among the machine memory banks. In this section, we present such functions and their functionalities. The functions are divided into five groups: control, architecture, data distribution, threads and statistic.

Control Functions: the system functions are functions to configure the interface. Such functions must be called in the begin (mai_init()) and end (mai_final()) of the application code. The function init() is responsible for



```
#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

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 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.

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

Architecture Functions: the architecture functions are used to retrieve information from the target architecture. Such functions help developers to better understand their platforms and, consequently, better manage memory affinity on their applications. The information retrieved by these functions are: node bandwidth, latency and bandwidth between two nodes, NUMA factor between two nodes, number of cores of a node. With such information, developers can decide which memory banks may be used to place data in order of optimize latency and bandwidth.

```
void mai_nodes(unsigned long nnodes, int *nodes);
double mai_nodebandwidth(int node);
double mai_numafactor(int node1, int node2);
double mai_bandwidth(int node1, int node2);
int mai_get_maxnodes();
int mai_get_maxcpus();
int mai_get_ncpusnode();
int* mai_get_cpusnode(int nodeid);
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();
```

Data Distribution Functions: these functions provides a simple way to divide, bind and get information about arrays on memory banks of a ccNUMA machine. They can help developers when creating their own memory policies. Functions responsible for array subdivision are named array_blocks functions. They can be use to cut array in sub-arrays or in sequences of bytes. Arrays are binded on memory banks using the bind functions. These functions can be used to bins blocks of an array on some nodes or to bind sequences of bytes in a selected memory bank of a node. Sometimes, it is also important to know what is the memory policy applied for an array and where memory pages of such array are placed. To get these information developers can use array information functions.

```
Array blocks:
void mai_subarray(void *p,int dim[]);
void mai_bytes(void* p,size_t bytes);
Data distribution:
int mai_regularbind(void *p);
int mai_irregularbind(void *p,size_t bytes,int node);
Array information:
void mai_mempol(void* ph);
void mai_arraynodes(void* ph);
```

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.

```
void mai_print_pagenodes(unsigned long *pageaddrs,int size);
int mai_number_page_migration(unsigned long *pageaddrs,int size);
double mai_get_time_pmigration();
void mai_get_log();
void mai_print_threadcpus();
int mai_number_thread_migration(unsigned int *threads,int size);
double mai_get_time_tmigration();
```

3.5 An Example of how to design a memory policy

In the section above, we present a set of functions that can be used to design memory policies inside MAi. In this section we present a simple example of how to use MAi system functions to create our own memory policy.

To design a memory policy, developers just have to write some functions that specify how data will be distributed among the ccNUMA machine memory banks. Such functions must be written using MAi system functions. Basically, developers must have to use architecture information and data distribution functions. The first set of functions allows developers to better understand the target platform and consequently better distribute data. The second ones provides different ways to divide/cut arrays into blocks and to bind such blocks on memory banks.

In the example (Figure 6), we present an implementation of a well know memory policy named First-touch (standard memory policy of Linux) that in this example we will name my_firsttouch. This memory policy distributes data among memory banks in a regular fashion (regular blocks sizes of data for each memory bank). Thus, all we have to do is divide data into blocks of the same size and then distribute such blocks on some memory banks considering the bandwidth between then.

In function my_firsttouch, the first thing that must be computed is the memory banks that will be used to place data. Functions mai_get_maxnodes, mai_bandwidth and mai_nodes allow developers to obtain the number of memory banks that are possible to be used, to get the bandwidth between such memory banks and to set the memory banks that will really used. After that, developers must specify how arrays will be divided by using, mai_bytes or mai_subarray. These two functions divide the array into blocks of the same size. In this example, the array was divided into blocks of same number of bytes, considering number of threads. Finally, we just have to tell MAi that we want to bind these blocks on the selected memory blocks. To do this, developers just have to call mai_regularbind function. This function will perform data alignment, data placement, compute available memory on the selected memory banks and data migration (if necessary).

Figure 7 shows the application source code that uses the memory policy described above. We can observe that before calling the memory policy, it is necessary to call mai_init function to initialize MAi library. Additionally, any array used by the memory policy may be allocated using MAi allocators. At this

```
 int* compute_distance(int nnodes)
{
  int i,j,find;
  float *band,aux,high;
  unsigned long *nodes;
  band = malloc((nnodes-1)*sizeof(unsigned long));
  nodes = malloc((nnodes)*sizeof(unsigned long));

  ......

  for(i=1;i<nnodes;i++)
   band[i-1]=mai_bandwidth(1,i+1);

  ......

  return nodes;
}

void my_firsttouch(void *array,int nthreads)
{
  int nnodes,*nodes;
   nnodes = mai_get_maxnodes();

  if(nthreads<=nnodes){
   nodes = compute_distance(nthreads);
   mai_nodes(nthreads,nodes);
  }
  else{
   nodes = compute_distance(nnodes);
   mai_nodes(nnodes,nodes);
  }

  mai_bytes(array,mai_getsize(array)/nthreads);
  mai_regularbind(array);
}
```

Figure 6: Example of MAi usage to develop memory policies

```
int main(int argc, char* argv[])
{
  // struct config information;
  double sum,b1,b2;
  int i,j,k,temp;
  size_t size;

  sum=b1=b2=0.0;

 omp_set_num_threads(atoi(argv[1]));

  mai_init(NULL);

  tab = mai_alloc_3D(X,Y,Z,sizeof(double),DOUBLE);
  my_firsttouch(tab,omp_get_num_threads());

#pragma omp parallel for default(shared) private(j,k)
    for(i=0;i<X;i++)
     for(j=0;j<Y;j++)
       for(k=0;k<Z;k++)
         tab[i][j][k]=0.0;

for(temp=0;temp<TEMP;temp++)
{
#pragma omp parallel for default(shared)
       private(j,k) firstprivate(sum,b1,b2)
  for(i=0;i<X;i++)
    for(j=0;j<Y;j++)
      for(k=0;k<Z;k++){
       sum = tab[i][j][k]* 2.0;
       tab[i][j][k]= sum * 2.0;
       sum++;
       b2=sum;
       b1 += tab[i][j][k];}
}

mai_final();

return 0;
}
```

Figure 7: The complete Source Code with the myfirsttouch Function

point, the memory policy can be called and data distribution will be performed by MAi. In the end of the application source code, the mai_function must be called to release allocated memory.

3.6 Implementation Details

MAi interface is implemented in C using Linux system calls. The interface support C/C++ applications developed for ccNUMA systems. Developers who wants to use MAi have to implement their applications using shared memory programming model (i.e, OpenMP, Posix Thread). To use this interface, ccNUMA platform must have a Linux version with support for NUMA architecture. Figure 8 present an overview of MAi on a ccNUMA subsystem.

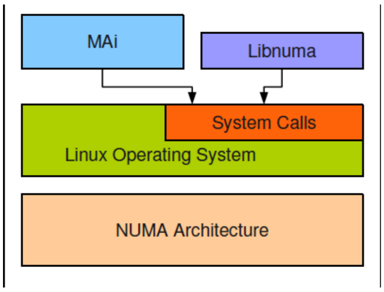


Figure 8: MAi Implementation

To guarantee memory affinity, all memory management is donne using MAi structures. These structures are loaded in the initialization of the interface (with the mai_init() function) and are used in the rest of application execution. All allocations and memory policies algorithms consider the NUMA architecture characteristics before perform any data allocation and placement on the memory banks.

As briefly discussed on section 3.3, MAi memory policies specifies how memory pages (what is the tile size) will be placed on the machine nodes (which nodes will be use and in which sequence). However, in order to optimize data distribution for ccNUMAs, MAi uses an strategy to decide the size of tiles and number of memory banks that will be used. The idea is: the tile size and memory banks usage depends on the number of threads running in the application. Additionally, MAi also considers ccNUMA characteristics when distributing data (latency and bandwidth). With these information, MAi can select the nodes and memory banks to use. For latency optimization data placement is attempted in placing data on adjacent memory banks. Next hop ccNUMA nodes will be used if and only if the number of threads is larger than the number of cores or the amount of memory is larger than the available memory on adjacent memory banks. For bandwidth optimization data placement is attempted in placing data on adjacent and next hop memory banks. In this case, a larger number of memory banks will be used in order to spread data among the machine memory banks. Using this strategy, MAi can reduce memory contention and load-balancing issues.

A memory policy set for an array of the application can be changed during the application steps. To do this developers just have to call the new memory policy function for the array and MAi will change it. In order to change memory policies, MAi verifies if memory banks have enough memory.

To better ensure memory affinity, both threads and memory must be considered in the solution. Due to this, MAi has a thread placement mechanism that is used for bind memory policies. The default thread placement 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 placement 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 ccNUMA platforms (e.g., SGI Altix Itanium 2 and Bull Novascale Itanium 2 NUMA platforms). Memory contention happens when several threads try to access the same memory bank at the same time. Concurrent accesses in the same memory bank can generate worse performance, since they must be serialized. The default placement strategy can be changed during the library initialization. The developer can then choose between using the operating system thread placement or defining his own threads and processors/cores mapping.

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 [30]. 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 9. 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).

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 9. 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-

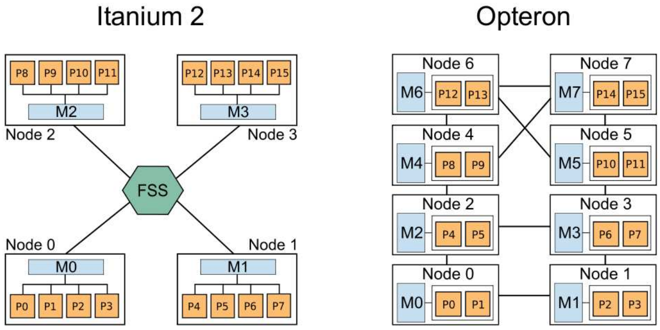


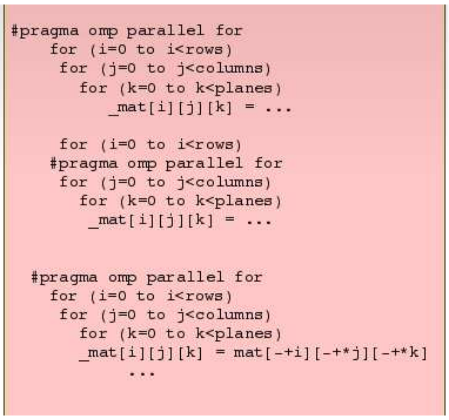
Figure 9: NUMA platforms.

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 [31]) and ICTM (Interval Categorization Tessellation model [32]). 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 10 present a schema of the application.



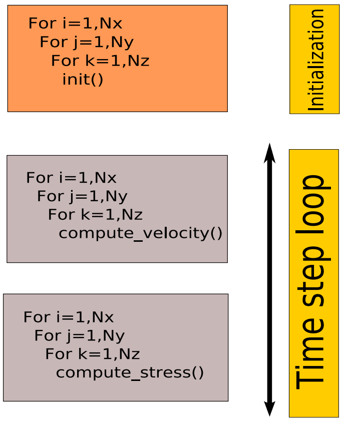
```
#pragma omp parallel for
    for (i=0 to i<rows)
     for (j=0 to j<columns)
       for (k=0 to k<planes)
          _mat[i][j][k] = ...

     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 10: 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 11 present a schema of the application.



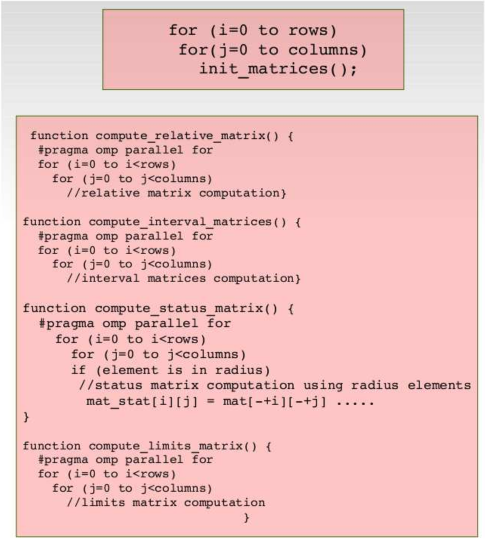
```
For i=1,Nx
 For j=1,Ny
  For k=1,Nz
   init()
```

```
For i=1,Nx
 For j=1,Ny
  For k=1,Nz
   compute_velocity()
```

```
For i=1,Nx
 For j=1,Ny
  For k=1,Nz
   compute_stress()
```

Figure 11: 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 12 present a schema of the application.



```
for (i=0 to rows)
 for(j=0 to columns)
   init_matrices();
```

```
 function compute_relative_matrix() {
  #pragma omp parallel for
  for (i=0 to i<rows)
    for (j=0 to j<columns)
      //relative matrix computation}

function compute_interval_matrices() {
  #pragma omp parallel for
  for (i=0 to i<rows)
    for (j=0 to j<columns)
      //interval matrices computation}

function compute_status_matrix() {
  #pragma omp parallel for
    for (i=0 to i<rows)
      for (j=0 to j<columns)
      if (element is in radius)
       //status matrix computation using radius elements
        mat_stat[i][j] = mat[-+i][-+j] .....
}

function compute_limits_matrix() {
  #pragma omp parallel for
  for (i=0 to i<rows)
    for (j=0 to j<columns)
      //limits matrix computation
                           }
```

Figure 12: 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 13(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.

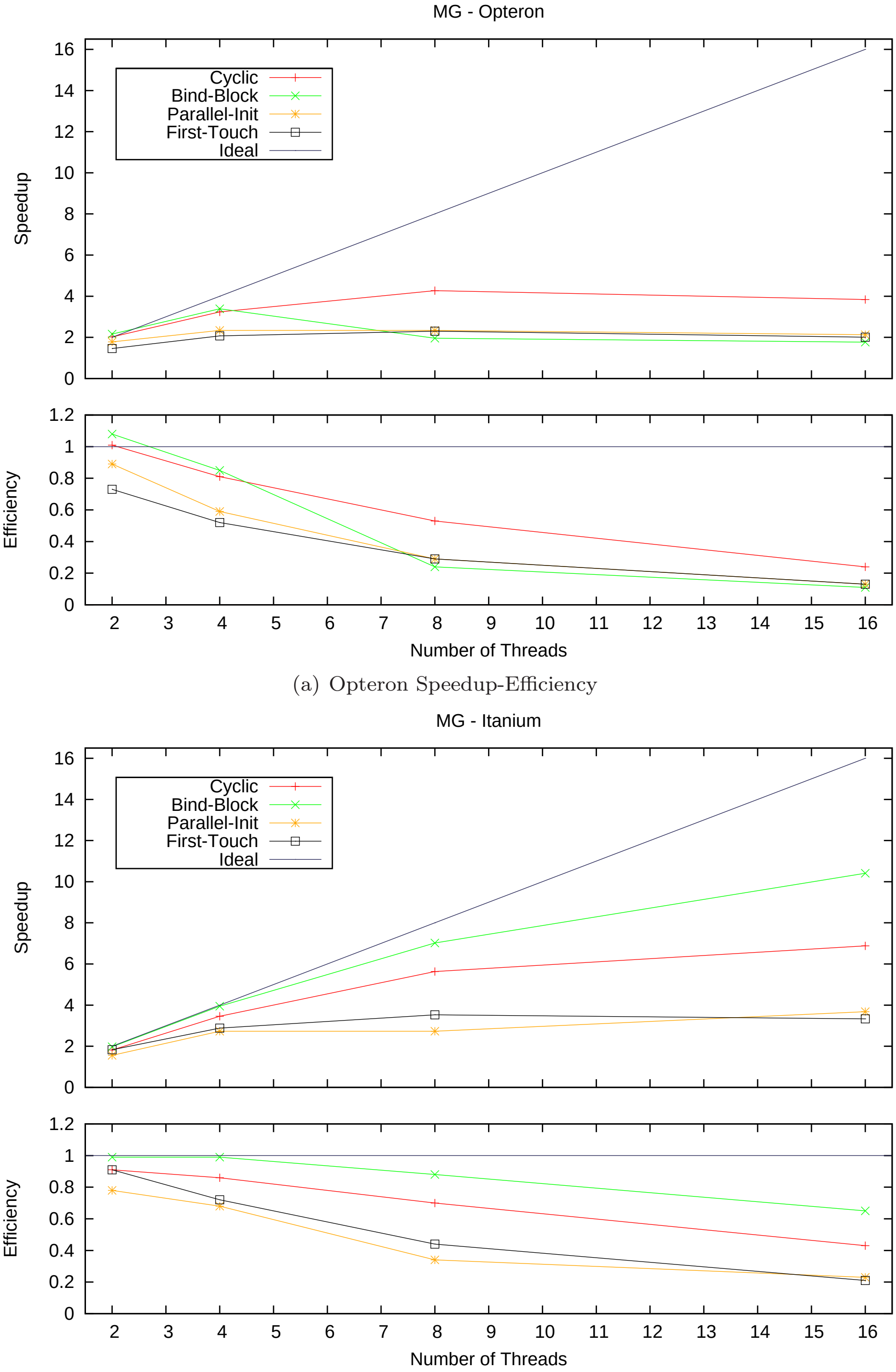
In figure 13(b) we present the speedups and efficiencies for the MG application for Itanium machine. The memory policy *bind_ block* is the best one, as this machine has a high NUMA factor try to allocate data close to threads that use it is better. The cyclic memory policy had worst results because 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 14(a) and 14(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 14(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.

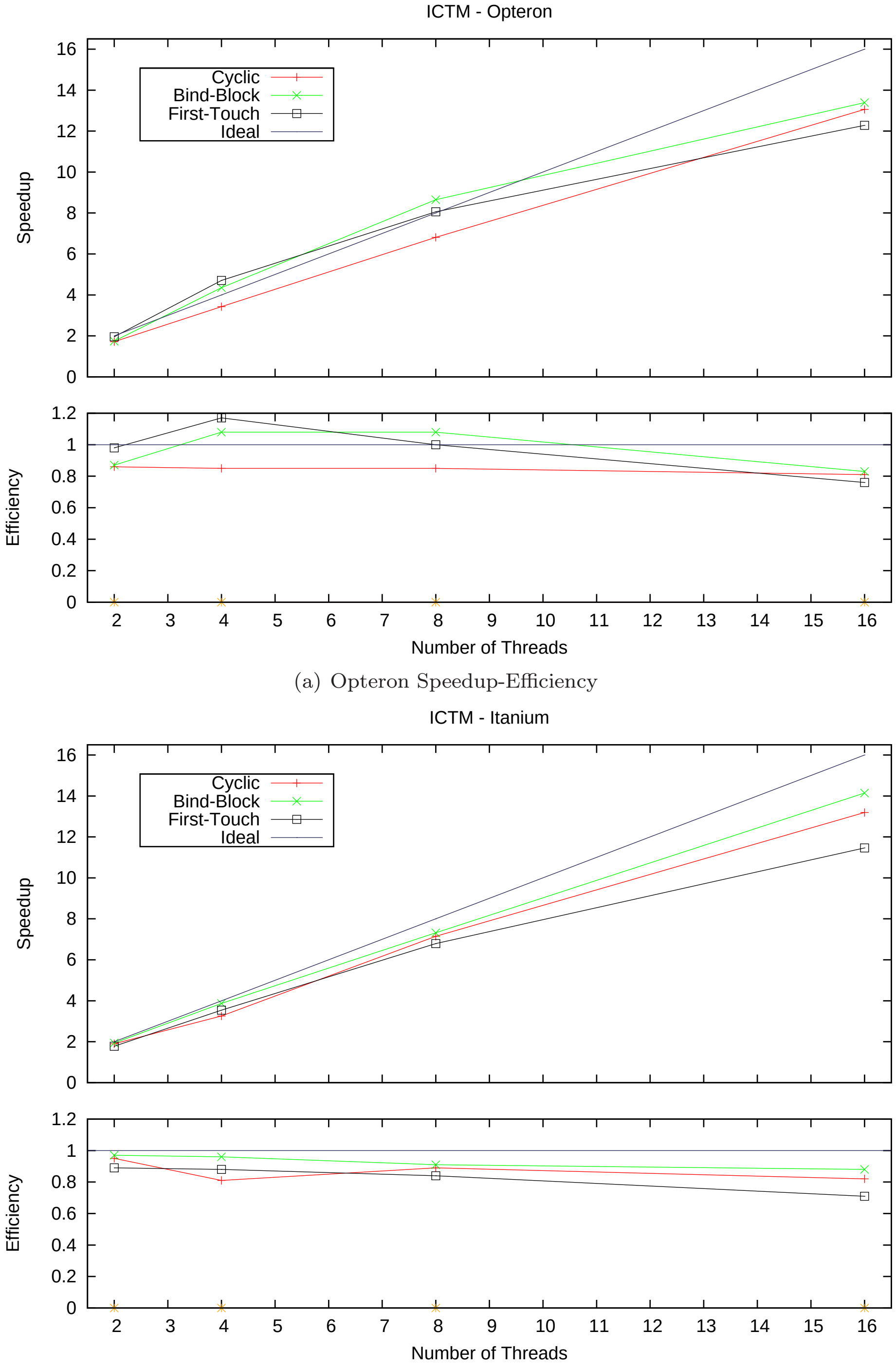
In figures 15(a) and 15(b) we present the results for Ondes 3D application considering the Opteron and the Itanium machines. As we can observe, the results obtained with MAi policies, First-Touch and Parallel-Init solutions were similar. The main reason for this is the regularity of data access patterns 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

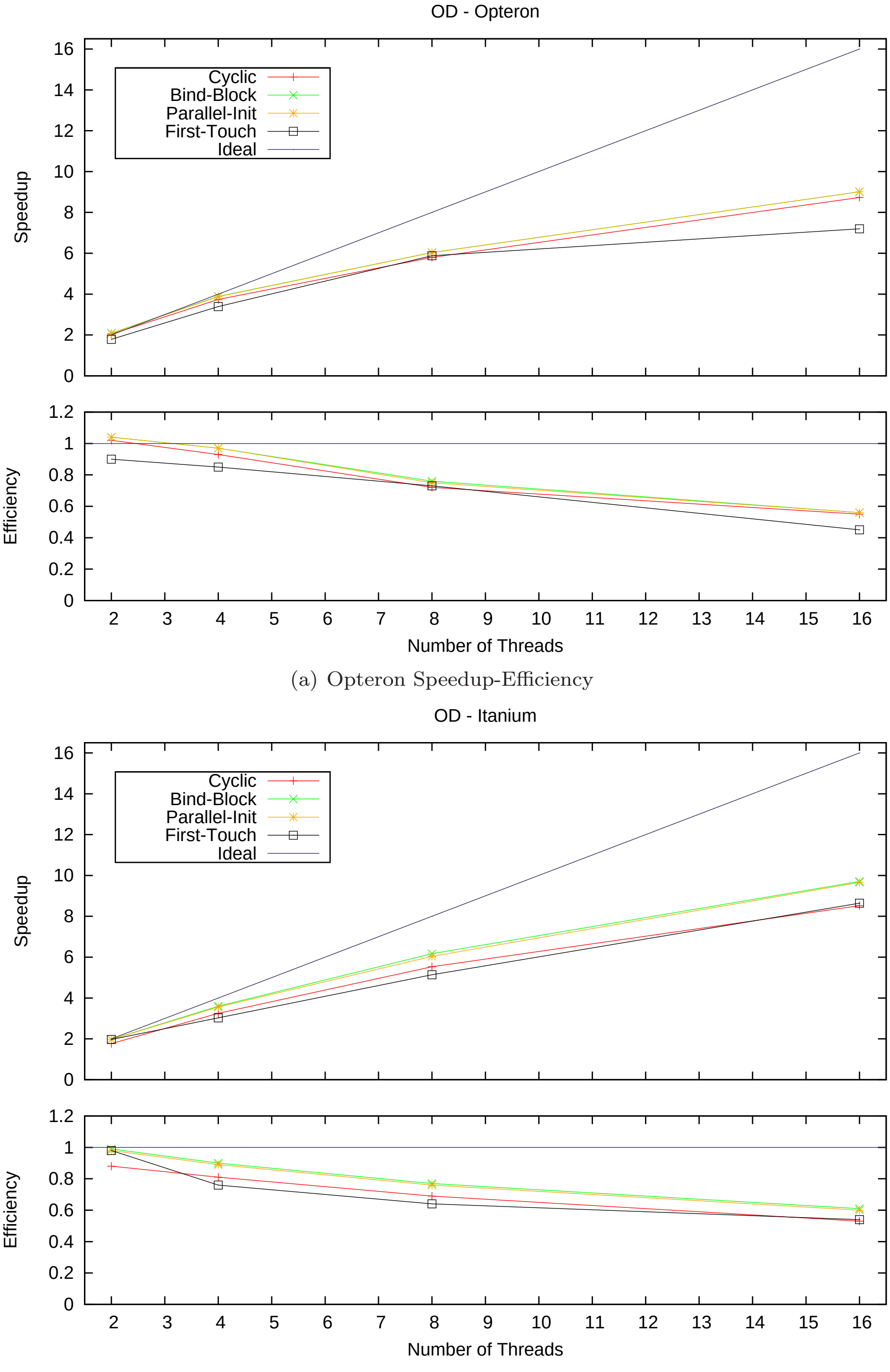
Figure 13: MG Application



(a) Opteron Speedup-Efficiency

(b) Itanium Speedup-Efficiency

Figure 14: ICTM Application



(a) Opteron Speedup-Efficiency

(b) Itanium Speedup-Efficiency

Figure 15: 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] I. C. Compiler, "Thread affinity interface," 2010. [Online]. Available: http://software.intel.com/en-us/intel-compilers/

[21] G. C. Compiler, "Thread affinity interface," 2010. [Online]. Available: http://gcc.gnu.org/onlinedocs/libgomp/Environment-Variables.html

[22] 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.

[23] H. Richardson, "High Performance Fortran: history, overview and current developments," Tech. Rep. TMC-261, 1996. [Online]. Available: http://hpff.rice.edu/publications/index.htm

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

[25] 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.

[26] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier, "Dynamic Task and Data Placement over NUMA Architectures: an OpenMP Runtime Perspective," in *5th International Workshop on OpenMP*. Dresden, Germany: Springer, 2009, pp. 79–92.

[27] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst, "Structuring the Execution of OpenMP Applications for Multicore Architectures," in *International Parallel and Distributed Symposium, IPDPS 2010*. Atlanta, United States: IEEE, 2010.

[28] 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/

[29] 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*. São Paulo, Brazil: IEEE, 2009.

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

[31] 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.

[32] 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