# Memory Affinity for Hierarchical Shared Memory Multiprocessors

Christiane Pousa Ribeiro*, Jean-François Méhaut*, Alexandre Carissimi†, Márcio Castro‡ and Luiz Gustavo Fernandes‡
*INRIA Mescal Research Team - LIG Laboratory
University of Grenoble, France
Email: {Christiane.Pousa, Jean-Francois.Mehaut}@imag.fr
†Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre, RS, Brazil
Email: asc@inf.ufrgs.br
‡GMAP - PPGCC - PUCRS
Porto Alegre, RS, Brazil
Email: {marcio.castro, luiz.fernandes}@pucrs.br

***Abstract*—Currently, parallel platforms based on large scale hierarchical shared memory multiprocessors with Non-Uniform Memory Access (NUMA) are becoming a trend in scientific High Performance Computing (HPC). Due to their memory access constraints, these platforms require a very careful data distribution. Many solutions were proposed to resolve this issue. However, most of these solutions did not include optimizations for numerical scientific data (array data structures) and portability issues. Besides, these solutions provide a restrict set of memory policies to deal with data placement. In this paper, we describe an user-level interface named Memory Affinity interface (MAi)[1], which allows memory affinity control on Linux based cache-coherent NUMA (ccNUMA) platforms. Its main goals are, fine data control, flexibility and portability. The performance of MAi is evaluated on three ccNUMA platforms using numerical scientific HPC applications, the NAS Parallel Benchmarks and a Geophysics application. The results show important gains (up to 31%) when compared to Linux default solution.**

***Keywords*-MAi; Menory Affinity; NUMA; NAS; ICTM.**

## I. Introduction

A Non-Uniform Memory Access (NUMA) platform is a hierarchical shared memory multiprocessed system in which the processing elements are served by multiple memory levels, physically distributed through the platform. Such distributed memory is seen by the developer as a single shared memory. Since memory is physically divided in blocks, the time spent to access a block is conditioned by the distance between the processor and the memory block (in which the data is physically allocated).

NUMA performance is essentially related to the affinity between threads and data. Memory affinity is the guarantee that memory access costs are reduced by either latency optimization or bandwidth increasing [1], [2]. In the last two decades, many researches have been carried out in the context of memory affinity, resulting in several proposals. These solutions were based in hardware counters, algorithms, Application Programming Interfaces (APIs) and operating systems supports that allow some memory control [3], [4], [5], [6], [7]. However, some problems remain: (i) they do not include optimizations for numerical scientific data (array data structures), (ii) they offer a limited set of memory policies, (iii) they do not allow fine data control and finally (iv) they do not offer platform/performance portability.

In this paper we present an user-level interface that deals with memory affinity for numerical scientific HPC applications in an efficient fashion, called Memory Affinity interface (MAi). It provides a wide set of memory policies to manage data allocation, distribution and access for scientific HPC applications based on shared memory programming model over Linux ccNUMAs. In order to evaluate its portability, flexibility and efficiency, we carried out experiments with some scientific memory-bound applications (NAS Parallel Benchmarks [8] and a Geophysics application [9]) on three ccNUMA platforms. The results are compared with one of the most used solution over ccNUMAs: the Linux default memory policy (*i.e., first-touch*).

This paper is organized as follows: in section 2, MAi characteristics and implementation details are presented. The performance evaluation of MAi is shown in Section 3. Section 4 discusses the related work. The concluding remarks and future works are pointed out in Section 5.

## II. MAi: Memory Affinity interface

MAi[2] is an user-level API that provides an efficient way for managing memory affinity on ccNUMA platforms for numerical scientific parallel applications. It simplifies memory affinity management issues, since it offers a wide set of memory policies to control data distribution. MAi main goals are: (i) flexibility of use (different policies in the same application), (ii) fine memory control (allows to express data access patterns), (iii) portability (once a

[1]Research supported by CAPES (grant number 4874-06-4) and French ANR - NUMASIS ANR-05-CIGC.

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

memory policy strategy is applied in the application source on a specific NUMA machine, it can be kept unchanged even if one must carry out experiments on another one) and (iv) performance gains (better performance than other solutions). In this section we describe the memory policies supported by MAi, the threads scheduling mechanism as well as its implementation details.

## A. Memory Policies

Before introducing the memory policies supported by MAi, we must first explain what the memory affinity unit of the interface is. In numerical scientific HPC, one cannot put aside the importance of arrays. Most of the algorithms are represented using these data structures. Due to this, in MAi, the memory affinity unit is an array. During MAi design, we have developped several optimizations for these data structures, since different arrays or even the same can have different access patterns during the application computation steps.

MAi implements seven memory policies that can be divided in three groups: bind, cyclic and random. The bind group is composed of *bind_all* and *bind_block* memory policies, the cyclic one of *cyclic*, *cyclic_block*, *skew_mapp* and *prime_mapp*, and the random one of *random* and *random_block*. The main differences between these three groups are the memory blocks used and data distribution. In MAi, data distribution can be performed using an array block (rows or columns).

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

In MAi, a block is a set of rows or columns that can be specified by the user. If the user does not specify the number of rows or columns, MAi will choose the block size automatically. The block size is computed considering the scheduling of the workload for the threads. This strategy is also applied for *cyclic* and *random* memory policies that use the concept of blocks. MAi *bind* policies have similar behavior of bind policy of the Linux NUMA solution. However, some modifications must be done in the application source code when applying the bind policy of the Linux NUMA solution over different platforms (e.g., reconfigure the bit masks in accordance with the platform number of nodes). This code portability is guaranteed by MAi and it is done transparently.

In Figure 1 (a) and Figure 1 (b), we show a schema that represents how data distribution is done in *bind_all* and *bind_block* memory policy. A node $n$ is composed of a memory block ($Mn$) and a set of processing units (to simplify memory policies representation, the processing units were not shown). Bind memory policies were designed for applications that present a regular behavior. In such applications, each thread always accesses the same set of data and a static scheduling of the workload is used. Furthermore, bind policies optimize latency over ccNUMAs, since data is placed closer to the thread that uses it.

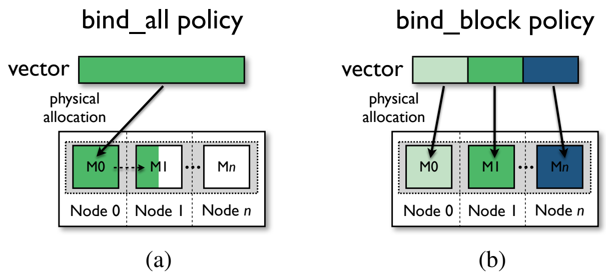


Figure 1. Bind memory policies.

The cyclic group uses different round-robin strategies to place data in the memory blocks of the platform. In both *cyclic* and *cyclic_block* policies, data is placed in the memory blocks in a linear round-robin way. First policy uses a memory page per round (Figure 2 (a)), which has similar behavior of the interleave policy of the Linux NUMA support, whereas the second one uses a block of memory pages (Figure 2 (b)).

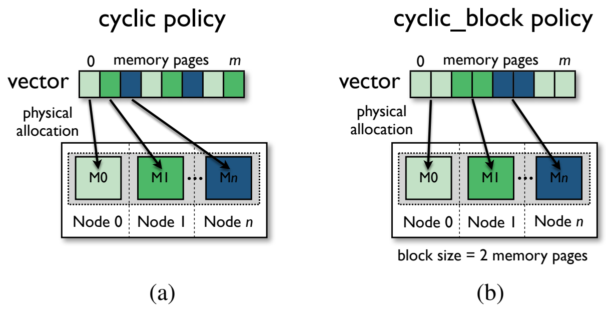


Figure 2. Cyclic memory policies.

The *skew_mapp* memory policy was proposed in [10] and it is a modification of round-robin policy that has linear page skew. In this policy, a page $i$ is allocated in the node $(i + \lfloor i/M \rfloor + 1)\ mod\ M$, where $M$ is the number of memory blocks (Figure 3 (a)). The *prime_mapp* policy was also proposed in [10] and uses a two-phase strategy. In the first phase, the policy places data using *cyclic* policy in $(P)$ virtual memory blocks, where $P$ is a prime greater or equal to $M$ (real number of memory blocks). In the second phase, the memory pages previously placed in the virtual memory blocks are reordered and placed into the real memory blocks also using the *cyclic* policy (Figure 3 (b)).

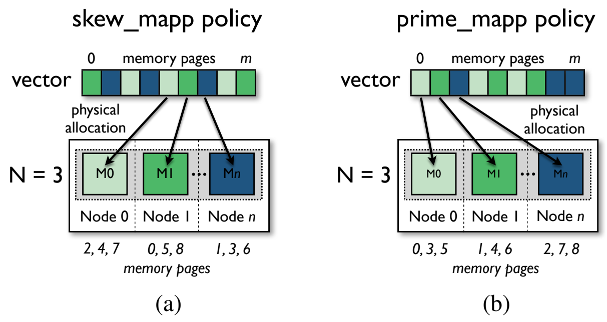


Figure 3. Cyclic memory policies.

Cyclic memory policies can be used in applications with regular and irregular behavior (threads do not always access the same data). These memory policies spread data between the memory blocks minimizing concurrent access and increasing bandwidth. However, some scientific applications can still have contention problems with *cyclic* and *cyclic_block*, since these policies make a linear distribution of memory pages (generally, power of 2) on the platform (it has power of 2 memory blocks). Thus, the proposal of *skew_mapp* and *prime_mapp* memory policies aims at reducing concurrent access for such applications [10].

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

One of the most important features of MAi is the fact that changing the memory policy applied to an array during the application execution is possible. 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. Arrays are usually accessed in different ways during the application life cycle. Thus, it is important to give the developer the opportunity to develop/create a memory policy suited for his application. Finally, any incorrect memory placement can be optimized through the use of the MAi memory migration functions. The unit used for migration can be a set memory pages (automatically defined by MAi) or a set of rows/columns (specified by user). However, migration must only be used to correct important data placement, since migration overheads can be high [11], [12], [5].

## B. Threads Scheduling

In some memory policies (bind group), to better ensure memory affinity, both threads and memory must be considered in the solution [13], [4], [6]. In MAi, for bind memory policies group, the default thread scheduling policy is to fix them to processors/cores. Using such a strategy assures that threads will not migrate (less overhead with task migrations) and that consequently, MAi will be able to perform a better data distribution and assure memory affinity. This thread scheduling considers the number of threads (T) and processors/cores (P) to decide how to fix threads. If $T \leq P$, one thread per processor/core strategy is chosen, which minimizes the memory contention problem inside the node, present in some NUMA platforms[3]. Memory contention happens when several threads try to access the same memory block. Concurrent accesses in the same memory block can generate worse performance, since they must be serialized. Since if $T > P$ MAi will schedule more threads per processor/core, scheduling one thread per processor/core can avoid this problem.

By default, MAi do not schedule threads for cyclic and random memory policies. Since these memory policies can be used in irregular applications (dynamic scheduling), not binding them to processors/cores and therefore letting the operating system control the threads scheduling will lead to better results. Additionally, the default scheduling strategy can be changed during the library initialization. The developer can then choose between using operating system thread scheduling or defining his own threads and processors/cores mapping.

## C. Implementation Details and Overview

MAi interface is implemented in C for Linux based ccNUMAs and it can be used in applications written in C or C++. In order to use MAi, applications must be developed using a shared memory programming model (specially, POSIX Threads or OpenMP). Since MAi uses some Linux NUMA system calls ($mbind()$, $move\_pages()$ and $migratepages()$), NUMA support (numaif.h) must be available in the Linux system.

All MAi functions are array-oriented and they can be divided in three groups: allocation, memory policies and system (Figure 4). 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 in memory blocks (as presented in section 2.1). Finally, with system functions, collecting and printing system information can be done (for instance memory blocks used by the memory policies, cpus/cores used during the application execution, memory blocks, statistics about page migration, etc.).

The interface uses a configuration file in which the user can describe the target NUMA platform specification (processors/cores and memory blocks to be used) and the thread scheduling that will be applied. If it is not specified,

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

**- Allocate an array of n dimensions**

```
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);
```

**- Apply cyclic memory policies - for arrays**

```
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);
```

**- Apply bind memory policies - for arrays**

```
void mai_bind_all(void *p);
void mai_bind_rows(void *p);
void mai_bind_columns(void *p);
```

**- Apply random memory policies - for arrays**

```
void mai_random(void *p);
void mai_random_rows(void *p, int nr);
void mai_random_columns(void *p, int nc);
```

Figure 4. MAi main functionalities.

the interface will collect the platform characteristics such as memory blocks, cpus/cores, caches sizes and NUMA factor, setting the configuration file automatically.

## III. Performance Evaluation

In this section we present the performance evaluation of MAi. First, we will describe the three NUMA platforms used in our experiments and then the numerical scientific HPC applications (NAS Parallel Benchmarks [8] and ICTM [9]) and their main characteristics. Finally, we will present the results and their analysis.

### A. NUMA Platforms

Our experiments were carried out in three ccNUMA platforms. The first platform is a sixteen Itanium 2 processors with 1.6 GHz each. It is organized in four nodes of four processors with 9 MB of shared L3 cache memory each. It has a total of 64 GB of main memory (16 GB of local memory). The nodes are connected using a FAME Scalability Switch (FSS), which is a backplane developed by Bull[4]. This connection gives different memory latencies for remote access by nodes (**NUMA factor from 2 to 2.5**[5]). The compiler that has been used for the OpenMP code compilation was the ICC (Intel C Compiler - version 9.0). A schematic figure of this machine is given in Figure 6 (a). We will use the name **Itanium2** for this machine.

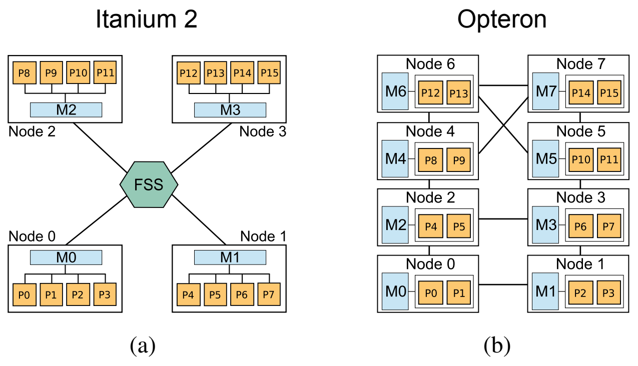


Figure 6. Itanium2 and Opteron platforms.

The second NUMA platform is an eight dual core AMD Opteron 2.2 GHz. It is organized in eight nodes of two processors with 2 MB of shared cache memory for each node. It has a total of 32 GB of main memory (4 GB of local memory). Each node has three connections which are used to link with other nodes (**NUMA factor from 1.2 to 1.5**). The compiler that has been used for the OpenMP code compilation was the GCC (GNU C Compiler). A schematic figure of this machine is given in Figure 6 (b). We will use the name **Opteron** for this platform.

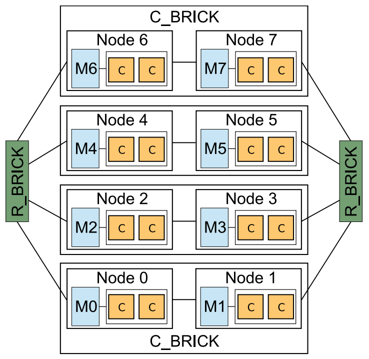
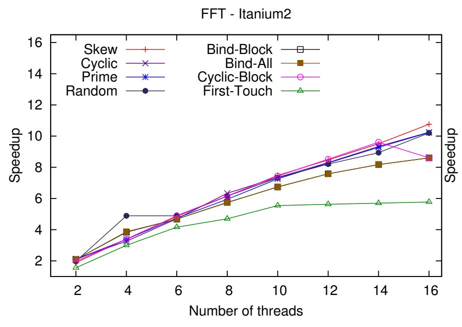


Figure 7. SGI NUMA platform.

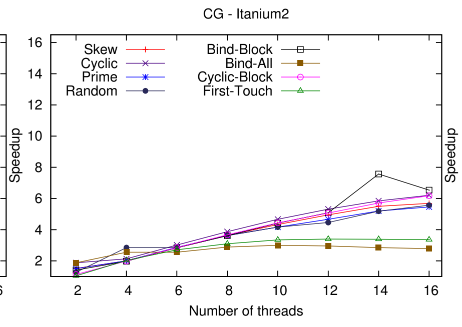
The last NUMA platform is an SGI Altix sixteen processors with 1.5 GHz and 4 MB of shared cache memory each. It is organized in eight nodes of two processors with a total of 32 GB of main memory (4 GB of local memory). Each node has two connections which are used to link with other nodes (**NUMA factor from 1.2 to 1.3**). The compiler that has been used for the OpenMP code compilation was the ICC (version 9.0). A schematic figure of this machine

[4]Bull - Architect of an Open World - http://www.bull.com

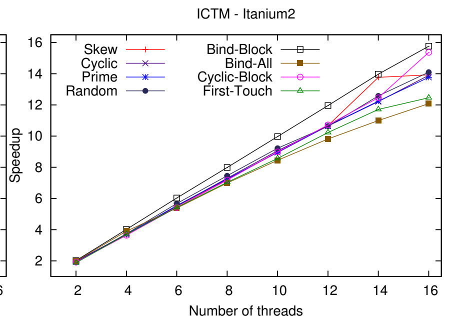
[5]NUMA factor is the ratio between remote memory access latency and local memory access latency.



(a)



(b)



(c)

Figure 5. Speedups for FFT (a), CG (b) and ICTM (c) on the Itanium 2 Platform.

is given in Figure 7. We will use the name **SGI** to make reference to this platform.

### *B. NAS Parallel Benchmarks*

NAS Parallel Benchmarks (NPB's) is a benchmark derived from computational fluid dynamics (CFD) codes and it is composed of applications and kernels [8]. From NPB's, we have selected two kernels: fast Fourier Transform (FFT) and Conjugate Gradient method (CG). Both kernels were chosen due to their memory access patterns (both have irregular data access patterns) and different data types (primitive C types and C structs). Additionally, they represent important classes of scientific computations. Such kernels were implemented in C using OpenMP to code parallelization.

FFT is a kernel that computes the fast transform of Fourier for three dimensional systems. The application works with complex numbers that are represented with structures. There are three main steps in the FFT computation and data are shared just in the second step. The computation is done in one direction step by step and each thread computes *Z* imaginary planes. In our experiments, we have used a $512x256x256$ matrix.

CG is also a kernel that uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel tests unstructured vector computations and communications. It uses a matrix with randomly generated locations of entries which gives a large amount of cache misses. The input parameter of this kernel is the size of the array that will be used for computation. In this case, we have used an array of size $75000$.

### *C. ICTM: Interval Categorizer Tessellation Model*

ICTM is a multi-layered tessellation model for the categorization of geographic regions considering several different characteristics (relief, vegetation, climate, etc.), using satellite images [14]. In order to categorize the regions, ICTM executes sequential phases in two dimensional matrices that are accessed in an irregular way. Each phase is responsible for computing data stored in different matrices. Since the categorization of extremely large regions has a high computational cost, a parallel solution for NUMA platforms was proposed in [9]. ICTM was implemented in C++ using OpenMP to code parallelization. In our experiments we have used matrices of size $6700x6700$ (2 GB of data).

### *D. Results*

For each application and architecture, we have carried out series of experiments using each of the seven memory policies of MAi (*bind_all*, *bind_block*, *skew_mapp*, *prime_mapp*, *cyclic*, *cyclic_block*, *random*) and the Linux default memory policy (*first_touch*).

The results presented for each experiment were obtained through the average of several executions varying the number of threads from 2 to 16, excluding the best and the worst execution times. These averages presented a low standard deviation, since all experiments have been done with exclusive access to the ccNUMA machines. The analysis of our results is based on nine distinct graphics, one for each platform/application. Each graphic compares the performances of all MAi memory policies with the *first-touch* policy.

In Figure 5, we present the speedups obtained with the eight memory policies for the three applications on Itanium 2 platform. Considering these speedups, we observe that the best results for ICTM (Figure 5 (c)) and CG (Figure 5 (b)) are obtained with MAi *bind_block* memory policy. By allocating data closer to the threads which computes them, we have decreased the number of remote accesses and consequently, the performance was increased. However, for FFT (Figure 5 (a)), the best memory policy was MAi *skew_mapp*. In this application, using memory policies that spread data among memory blocks will lead to better performance, since the data access pattern is irregular (threads use different sets

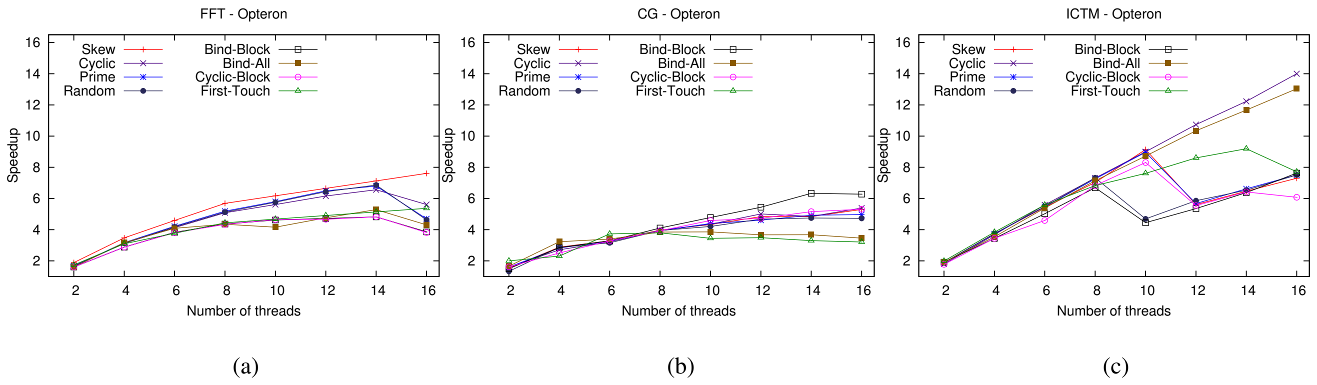


Figure 8. Speedups for FFT (a), CG (b) and ICTM (c) on the Opteron Platform.

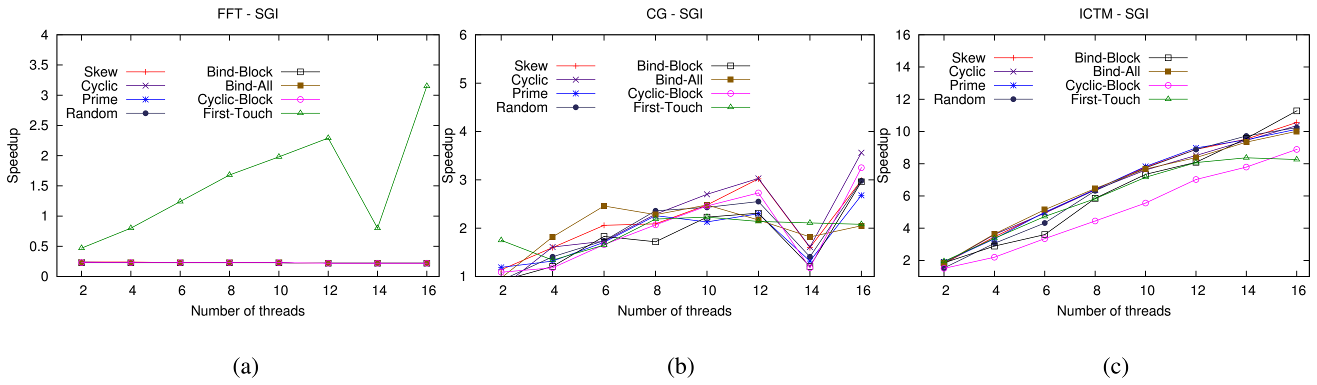


Figure 9. Speedups for FFT (a), CG (b) and ICTM (c) on the SGI Platform.

of data during the application steps).

Figure 5 also shows that Linux default memory policy (*first_touch*) presented one of the worst results (in some cases they were better than *bind_all*). *First_touch* policy was designed to optimize latency, since it places memory pages once (when they are first touched) and never migrate them. If some thread needs a memory pages different from the first page that it touched (e.g., share some data), it will have a remote access. Thus, it only presents interesting results in applications that have regular access patterns. However, in FFT, CG and ICTM, threads usually access different data sets, so this policy does not result in a good performance.

Since Itanium 2 has a high NUMA factor, which gives expensive remote accesses, the best memory policies for this platform are the ones designed to bind data closer to the threads that use them. That is the case of MAi *bind_block* memory policy, which optimizes latency by reducing the number of remote accesses and NUMA effects.

In Figure 8, Opteron results that, on average, the best speedups for FFT, CG and ICTM were obtained with the memory policies that avoid the contention problem (*skew_mapp*, *cyclic*, *random*) and increase the network bandwidth. The other memory policies resulted in worse speedups, since they prioritize latency optimizations (which must not be the main concern in this platform). We can also observe that the *first_touch* policy presents interesting speedups. However, the speedups for this policy were 15% lower than MAi policies. *First_touch* is indicated for regular applications over platforms that have high NUMA factors, which is not the case in this platform (its main characteristics are bandwidth problems and a small NUMA factor). As a consequence, remote accesses are not very expensive but contention in memory blocks can still occur. Due to this fact, memory policies designed to spread data among the platform memory blocks present better results. By distributing memory pages in a cyclic/random way, memory contention and consequently, the influence of network bandwidth problems can be avoided.

In Figure 9, we present the speedups obtained on SGI platform. MAi *cyclic* and *bind_block* policies were the best solutions for CG and ICTM applications over SGI. However, the difference between MAi best policies and *first_touch* was not so high due to the network interconnection constraints (for instance small NUMA factor). We can conclude that FFT ran slower with MAi memory policies and did not have speedups (in this case, *First_touch* presented the best results). For FFT, we have applied MAi memory policies only in the arrays, since MAi is array-oriented whereas *First_touch* policy was applied for all application data segment. Another problem remained concerning the memory contention in platform, since the matrices of complex numbers can be accessed in different steps by different threads. For this application, more experiments must be performed to better understand that behavior.

The SGI NUMA platform has, as its main characteristics, the fat-tree network and a small NUMA factor. This hierarchical network connection has two levels: the first level is inside the C-Brick (smaller latency) and the second level is outside the C-Brick (higher latency). Thus, remote accesses will only be expensive when threads access data outside the C-Brick (Figure 9). In this platform, applications that deal with unstructured and sparse matrices (e.g., CG) will have better speedups when memory policies that spread data among platform memory blocks are applied.

In High Performance Computing (HPC), platforms are generally used in their full capacity in order to reach their maximum efficiency. Because of that, we present in Table I the highest efficiency (with 16 threads) of MAi best memory policy for each application/platform in comparison to the *first_touch* policy (FT).

Table I
EFFICIENCY OF MAI BEST POLICY (MAI) AND FIRST-TOUCH (FT).

| | Itanium 2 | | Opteron | | SGI | |
|---|---|---|---|---|---|---|
| | MAi | FT | MAi | FT | MAi | FT |
| FFT | 67.24% | 36.12% | 47.54% | 33.57% | 1.38% | 19.69% |
| CG | 40.36% | 21.03% | 39.28% | 20.06% | 22.25% | 12.19% |
| ICTM | 98.51% | 77.89% | 60.18% | 48.17% | 66% | 51.61% |

Considering Itanium2 NUMA platform, MAi memory policies improved the efficiency of FFT, CG and ICTM by 31%, 19% and 21% respectively, compared to the *first_touch* memory policy. On Opteron, MAi memory policies improved the efficiency of FFT, CG and ICTM by 14%, 19% and 12% respectively. Finally, MAi did not present high improvements in terms of efficiency on SGI with CG and ICTM (10% and 15% respectively) in comparison to *first_touch*. For FFT, *first_touch* memory policy were 18% better than the best MAi policy. As mentioned before, more experiments will be done to better explain this behavior.

## IV. RELATED WORK

In scientific HPC, the assurance of memory affinity on ccNUMA platforms is an important issue. Research groups have studied different ways to manage memory affinity on Linux based NUMA platforms [15]. As a result, libraries/interfaces [7] and design of memory policies in user or kernel spaces of operating systems have been proposed [6], [12], [4], [5].

Linux operating system has a basic support to manage affinity on ccNUMAs. This support has three parts: kernel/system calls, a library (libnuma) and a tool (numactl). The kernel part defines three system calls ($mbind()$, $set\_mempolicy()$ and $get\_mempolicy()$) that allow the programmer to set a memory policy (bind, interleave, preferred or default) for a memory range. However, the use of such system calls is a complex task, since developers must deal with pointers, memory pages, sets of bytes and bit masks. The numactl tool allows the user to set a memory policy for an application without changing the source code. However, the chosen policy will be applied over all application data (it is not possible to either express different access patterns or change the policy during the execution [7]). The last part of this support is a library named libnuma, which is a wrapper layer over the kernel system calls. The limited set of memory policies provided by libnuma is the same as the one provided by the system calls.

The Linux NUMA support does not have portability, since source code that manage memory affinity must be changed for different architectures. MAi avoids this necessity of source code modifications since it collects the architecture characteristics automatically. Moreover, after analyzing the experimental results, we concluded that a wide set of memory policies is necessary, since different scientific HPC applications and ccNUMA platforms demand different memory policies (latency or bandwidth optimizations) to assure memory affinity.

Linux operating system implements *first-touch* as default policy to manage memory affinity on ccNUMAs. This policy places data in the node that first accesses it [4], [15]. To improve memory affinity using this policy, it is necessary to either execute a parallel initialization of all application data allocated by the master thread or allocate its on data on each thread. However, this strategy will only present performance gains if it is applied on applications that have a regular data access pattern. In case of irregular applications, *first-touch* will result in a high number of remote memory accesses, since threads do not access the same data. Our solution overcomes these issues offering cyclic and random memory policies and data migration functions.

In [6], [12], [5], the authors have designed and implemented the *on-next-touch* memory policy on Linux operating systems. This policy allows data migration when threads touch them for the next time, allowing more local accesses. Its performance evaluation has shown good performance gains only for applications that have a single level of parallelism. When it was applied in nested parallel levels, it was not profitable (threads frequently lost their affinity). Thus, many data migrations were done and this overhead lowered the performance gains. MAi avoids it by combining static data distribution with small number of migrations.

## V. Conclusions and Future Work

We have focused our work on MAi, a memory affinity interface to manage memory placement on ccNUMA platforms for scientific HPC applications based on arrays. We also have presented its proposal, its main functionalities, its implementation details and advantages (fine data control, flexibility and portability). In order to evaluate its performance, we have carried out experiments over three ccNUMA platforms using NPB's and a Geophysics application.

We have observed a performance improvement up to 31% in relation to the *first-touch* (FFT kernel over Itanium 2 platform). Gains were also observed for all applications over the three ccNUMA platforms used in our experiments. In most of the experiments, MAi memory policies presented better results than *first_touch* policy. However, due to a lack of performance in FFT kernel, MAi interface still needs to be improved for SGI platform.

The results have also shown that different ccNUMAs platforms and applications need different memory policies. ccNUMAs platforms with a high NUMA factor demand memory policies such as *bind_block*, whereas cyclic memory policies present more satisfying results in platforms with bandwidth problems. Performance can be improved in applications with regular data access patterns by using *bind_block* memory policies, since threads always access the same data set. For irregular applications, the use of *cyclic* or *random* memory policies usually results in higher performance gains, since there is no specific data access pattern in such applications (it can be changed during the execution).

The implementation of memory policies using the concept of plug-ins, hierarchical tiles for 3D/4D arrays, the performance evaluation over several applications and MAi integration in a compiler preprocessor are issues to be considered in our future works.

## References

[1] 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

[2] C. Ribeiro, F. Dupros, A. Carissimi, V. Marangozova-Martin, J.-F. Méhaut, and M. S. de Aguiar, "Explorando Afinidade de Memória em Arquiteturas NUMA," in *WSCAD '08: Proceedings of the 9th Workshop em Sistemas Computacionais de Alto Desempenho - SBAC-PAD*. Campo Grande, Brazil: SBC, 2008.

[3] 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.

[4] 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

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

[6] 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

[7] 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

[8] 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

[9] 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.

[10] 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.

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

[12] C. Terboven, D. A. Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and Thread Affinity in OpenMP Programs," in *MAW '08: Proceedings of the 2008 workshop on Memory access on future processors*. New York, NY, USA: ACM, 2008, pp. 377–384. [Online]. Available: http://dx.doi.org/10.1145/1366219.1366222

[13] F. Bellosa, "Memory Conscious Scheduling and Processor Allocation on NUMA Architectures," Tech. Rep. TR-I4-95-06, Aug. 1995. [Online]. Available: http://i30www.ira.uka.de/

[14] M. S. de Aguiar, G. P. Dimuro, and A. C. da Rocha Costa, "ICTM: An Interval Tessellation-Based Model for Reliable Topographic Segmentation," *Numerical Algorithms*, vol. 37, no. 1–4, pp. 3–11, 2004.

[15] 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.