NUMA-ICTM: A Parallel Version of ICTM Exploiting Memory Placement Strategies for NUMA Machines

Márcio Castro, Luiz Gustavo Fernandes
GMAP, PPGCC
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre - Brazil
{mcastro, gustavo}@inf.pucrs.br

Christiane Pousa, Jean-François Méhaut
Laboratoire d'Informatique Grenoble
Grenoble Université
Grenoble - France
{christiane.pousa, mehaut}@imag.fr

Marilton Sanchotene de Aguiar
GMFC, PPGInf
Universidade Católica de Pelotas
Pelotas - Brazil
marilton@atlas.ucpel.tche.br

Abstract

In geophysics, the appropriate subdivision of a region into segments is extremely important. ICTM (Interval Categorizer Tesselation Model) is an application that categorizes geographic regions using information extracted from satellite images. The categorization of large regions is a computational intensive problem, what justifies the proposal and development of parallel solutions in order to improve its applicability. Recent advances in multiprocessor architectures lead to the emergence of NUMA (Non-Uniform Memory Access) machines. In this work, we present NUMA-ICTM: a parallel solution of ICTM for NUMA machines. First, we parallelize ICTM using OpenMP. After, we improve the OpenMP solution using the MAI (Memory Affinity Interface) library, which allows a control of memory allocation in NUMA machines. The results show that the optimization of memory allocation leads to significant performance gains over the pure OpenMP parallel solution.

1. Introduction

An adequate subdivision of geographic areas into segments presenting similar characteristics is often convenient in Geophysics. This appropriate subdivision enables us to extrapolate the results obtained in some locations within the segment, in which extensive research have been done, to other locations less explored within the same segment. Thus, we can have a good understanding of the locations which have not been thoroughly analyzed [3].

ICTM (Interval Categorizer Tessellation Model) is a tessellation model for the simultaneous categorization of geographic regions considering several different characteristics (relief, vegetation, climate, land use, etc.) using information extracted from satellite images. The analysis of the function monotonicity, which is embedded in the rules of the model, categorizes each tessellation cell, with respect to the whole considered region, according to its declivity signal (positive, negative or null). The first formalization of ICTM, a single-layered model for the relief categorization of geographic regions, called Topo-ICTM (Interval Categorizer Tessellation Model for Reliable Topographic Segmentation), was initially presented in [4]. Through this work, it was possible to find out that the categorization of large regions requires a high computational power, resulting in large execution times over single processor machines.

Previous works investigated the possibility to parallelize ICTM using distributed memory platforms such as clusters and grids (see Section 2.2), however these platforms introduce two important limitations to the ICTM parallelization: (i) they do not allow parallel approaches which need intensive processes communication, since the communication cost is too significant, and (ii) such platforms usually do not present nodes with large local memories, which are necessary to compute very large regions.

Traditional UMA (Uniform Memory Access) architectures present a single memory controller, which is shared by all processors. This single memory connection often becomes a bottleneck when many processors accesses the memory at the same time. This problem is even worse in systems with a higher number of processors, in which the single memory controller does not scale satisfactorily. Therefore, these architectures may not fulfill our requirements to develop an efficient parallel solution for ICTM.

NUMA (Non-Uniform Memory Access) architectures appear as an interesting alternative to surpass the UMA scalability problem. In NUMA architectures the system is split into multiple nodes [6]. These machines have, as their main characteristics, multiple memory levels that are seen by the developers as a single memory. They combine the efficiency and scalability of MPP (Massively Parallel Processing) architectures with the programming facility of SMP (Symmetric Multiprocessing) machines [9]. However, due to the fact that the memory is divided in blocks, the time spent to access the memory is conditioned by the "distance" between the processor (which accesses the memory) and the memory block (in which the data is allocated).

A parallel solution of ICTM for NUMA machines exploiting memory affinity in order to achieve better performances is the aim of in this paper. First, we describe how ICTM was parallelized using OpenMP. After that, considering the fact that OpenMP has been originally developed to parallelize applications for UMA machines, we chose the MAI (Memory Affinity Interface) library in order to control de memory allocation and threads placement.

This paper is structured as follows: Section 2 describes the general workflow of ICTM and the related work about ICTM parallel versions for other high performance platforms. In Section 3, we briefly present how ICTM was parallelized using just OpenMP library, we describe the machines used to run our experiments and the case studies used to evaluate its performance. In order to face the pure OpenMP solution limitations, in Section 4 we introduce the MAI functionalities to fine tune memory allocations. Finally, concluding remarks and future works are pointed out in Section 5.

2. ICTM

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.). The number of characteristics that should be studied determines the number of layers of the model. In each layer, a different analysis of the region is performed. An appropriate projection of all layers to a basic layer of the model leads to a meaningful subdivision of the region and to a categorization of the sub-regions that consider the simultaneous occurrence of all characteristics, according to some weights, permitting interesting analyses about their mutual dependency.

The input data is extracted from satellite images, in which the information is given in certain points referenced

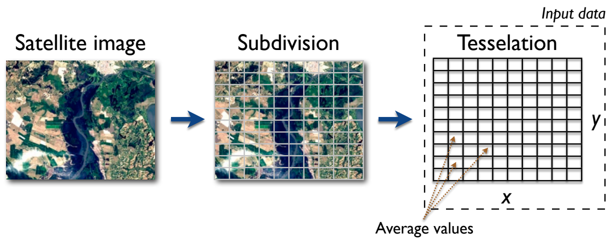

Figure 1. ICTM input data.

by their latitude and longitude coordinates. The geographic region is represented by a regular tessellation that is determined by the subdivision of the total area into sufficiently small rectangular subareas, each one represented by one cell of the tessellation (Figure 1). This subdivision is done according to a cell size, established by the geophysics or ecology analyst and it is directly associated to the refinement degree of the tessellation.

2.1. Categorization Process

In order to categorize the regions of each layer, ICTM executes sequential phases, where each phase uses the results obtained from the previous one (Figure 2). The tesselation showed in Figure 1 is represented as a matrix with n_r rows and n_c columns.

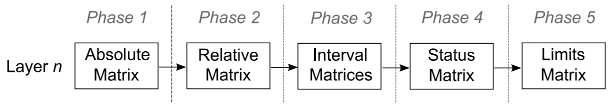

Figure 2. ICTM categorization process.

In topographic analysis, usually there are too many data, most of which is geophysically irrelevant. Thus, for each subdivision, the average value of a specific feature at the points supplied by radar or sattelite images is taken. The first phase of the categorization process involves this input data reading (average values) and these data are stored on a matrix called **Absolute Matrix**.

The categorization proceeds to the next phase, in which the data is simplified. The Absolute matrix is normalized by dividing each element by the largest one, creating the **Relative Matrix**. Considering the fact that the data that has been extracted from the satellite images are very accurate, the errors contained in the Relative Matrix come from the discretization of the region into tessellation cells. Due to this fact, Interval Mathematics techniques [8] are used to control the errors associated to cell values (advantages of using intervals can be seen in [3] and [5]). Thus, in the next phase, two **Interval Matrices** are created in which the interval values for x and y coordinates are stored.

The most important phase of the entire process is the creation of the **Status Matrix**. In this phase, each cell is compared to its neighbors in four directions. For each cell, four directed declivity registers – $reg.e$ (east), $reg.w$ (west), $reg.s$ (south) and $reg.n$ (north) – are defined, indicating the admissible declivity sign of the function that approximates it in any of these directions, taking into account the values of the neighbors cells. The number of neighbors to be analysed in each direction is a parameter called **radius**.

For non-border cells: $reg.X = 0$, if there exists a non-increasing approximation function between the cell and its neighbors at X direction; $reg.X = 1$, otherwise. For east, west, south and north border cells $reg.e = 0$, $reg.w = 0$, $reg.s = 0$ and $reg.n = 0$, respectively.

Let $w_{reg.e} = 1$, $w_{reg.s} = 2$, $w_{reg.w} = 4$ and $w_{reg.n} = 8$ be weights to be associated to the directed declivity registers. The status matrix is defined as an $n_r \times n_c$ matrix where each entry is the value of the corresponding cell state, calculated as the value of the binary encoding of the corresponding directed declivity registers, given as $status_{cell} = (1 \times reg.e) + (2 \times reg.s) + (4 \times reg.w) + (8 \times reg.n)$. Thus, for a given cell, the correspondent cell can assume one and only one state represented by the value $status_{cell} = 0..15$.

In the last phase, the **Limits Matrix** is created. A limit cell occurs when the function changes its declivity, presenting critical points (maximum, minimum or inflection points). To identify such limit cells, a limit register associated to each cell is used. The border cells are assumed to be limit cells.

The categorization of extremely large regions has a high computational cost. Its cost is basically related to two parameters: the matrices number of cells and the number of neighbors that are analyzed during the categorization process in each layer (radius).

2.2. Related Works

In [12], the authors have presented a parallel version of ICTM for clusters. In that work, the authors have used the master-slave model to compute layers in parallel, since there is no data dependence between layers. However, considering the fact that each slave process calculates a given layer of the model, the maximum size of each layer is limited by the memory size of the node in which the slave process is running. As a consequence, very large regions can not be categorized using this kind of decomposition method.

On the other hand, in [11] the authors have shown a parallel version of ICTM for grids. This paper presents two different ways to parallelize the ICTM model: using centralized or distributed data. The second solution is more appropriated for grids, since it reduces drastically the communication between computing nodes. Nevertheless, the data must be previously stored in each machine of the grid.

In brief, these two previous solutions have presented different ways to parallelize ICTM, showing interesting results. However, they have their limitations, specially when the categorization of extremely large regions is required. In this scenario, shared memory architectures appear as an attractive alternative to achieve better results. Additionally, the specific utilization of NUMA machines allows the use of a larger number of processors.

3. ICTM Parallelization with OpenMP

OpenMP is a standard widely used API (Application Programming Interface) for the development of parallel applications for shared memory environments [13]. It has been developed for UMA architectures and it does not make any assumptions about the physical location of data in memory or threads [13, 13]. Aiming to solve this problem, several extensions of the standard OpenMP for NUMA architectures were proposed [2, 1, 7]. However, none of these extensions became a standard.

In this work, we have used OpenMP to parallelize ICTM. We have chosen OpenMP because of its simplicity, since the sequential code can be parallelized with few modifications. One of its main advantages is that any operation of creation/destruction of threads is done transparently by the API. Moreover, OpenMP uses the fork-join model, which allows the existence of several sequential and parallel regions in the source code. This model can be easily used with the ICTM sequential code, allowing the parallelization inside each step of the categorization process.

3.1. Parallel Approach

The ICTM categorization process can be basically divided into two parts. The first part is the data initialization, in which the information read from the satellite image is written among the Absolute Matrix cells. The second part is the categorization and it is composed by several other phases. In this paper we focus on the second part, since it is the most computational intensive one.

As mentioned before, each phase executes some computation over all cells of its respective matrix, modifying their values. In a simplified way, it was implemented using a two **nested for loops** structure. So, it is possible to use the omp parallel for directive inside each phase to distribute the work among the threads. Thus, each thread will compute a subset of the respective matrix rows, as follows:

```
#pragma omp parallel for
for (i = 0; i < rows); i++)
  for (j = 0; j < columns); j++)
    //computation
```

The *pragma* directive is responsible for threads creation, work distribution among them and threads destruction (after the end of the computation). We believe that this is a simple and elegant solution, since we have not done many changes in the sequential source code. However, OpenMP directives do not allow to control memory allocation among the NUMA nodes and threads migration. Those procedures are done according to the Linux kernel policies.

3.2. Performance Evaluation

In this section we present the performance evaluation of the parallel ICTM using the OpenMP. We describe two NUMA platforms and the case studies we have used in our experiments to evaluate the OpenMP solution. The reason for the use of two different NUMA machines is to evaluate the impact of different NUMA factors [1] in the choice of memory allocation strategies.

3.2.1 NUMA Machines

The first NUMA machine is an eight dual core AMD Opteron 2.2 GHz and 2 MB of cache memory for each processor. It 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 3. From now onwards, we will use the name **Opteron** for this machine.

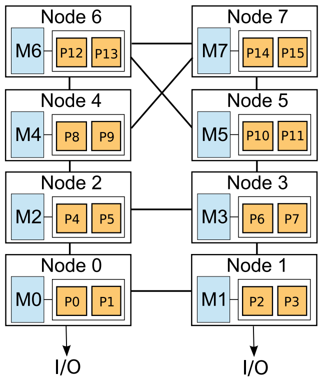

Figure 3. AMD Opteron machine.

The operating system is the Debian distribution of Linux version 2.6.23-1-amd64 with NUMA support (system calls and user API numactl). The compiler for the OpenMP code compilation is the GNU Compiler Collection (GCC).

[1]The NUMA factor is obtained through the division of remote latency by local latency.

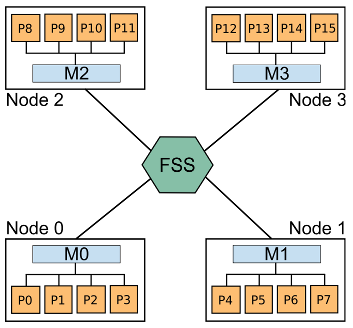

Figure 4. Itanium 2 machine.

The second NUMA machine used is a sixteen Itanium 2 with 1.6 GHz and 9 MB of L3 cache memory for each processor. It is organized in four nodes of four processors each and has in total 64 GB of main memory. This memory is divided in four blocks for each node (16 GB of local memory). Nodes are connected using a FAME Scalability Switch (FSS). 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 4. From now onwards, we will use the name **Itanium 2** for this machine.

The operating system is the Red Hat distribution of Linux version 2.6.18-B64k.1.21 with NUMA support (system calls and user API numactl). The compiler for the OpenMP code compilation is the ICC (Intel C Compiler version 9.1.045).

3.2.2 Case Studies

The case studies have been chosen in terms of memory usage and computing power necessity. Considering the total amount of memory in both NUMA machines, we have selected four sizes of matrices. Moreover, in order to comprehend the influence of the radius in our parallel solution, we have done experiments with three different values of radius. These case studies are shown in Table 1 and they have been used to measure the overall performance of our solution.

Table 1. Case studies.

Name	Size of matrices	Memory usage	Radius
Case 1	4,800x4,800	1 GB	20, 40, 80
Case 2	6,700x6,700	2 GB	20, 40, 80
Case 3	9,400x9,400	4 GB	20, 40, 80
Case 4	13,300x13,300	8 GB	20, 40, 80

The results presented for each case study in Sections 3.2.3 and 4.3 were obtained through the average of 10 executions, 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 NUMA machines.

3.2.3 OpenMP Results

In this section we show the results that we have obtained with both Opteron and Itanium 2 NUMA architectures. First, we have fixed the matrix size in order to show how the OpenMP parallel solution behaves varying the radius. After, we have fixed the radius to compare the speed-ups varying the matrices sizes according to the case studies. The chosen matrix size and value of radius for these experiments were respectively dimension 6,700 (Case 2) and 40. According to a previous analysis of the obtained results, we have noticed that this configuration presents the best balance between the input image size and the level of details required for a useful analysis in terms of computational cost.

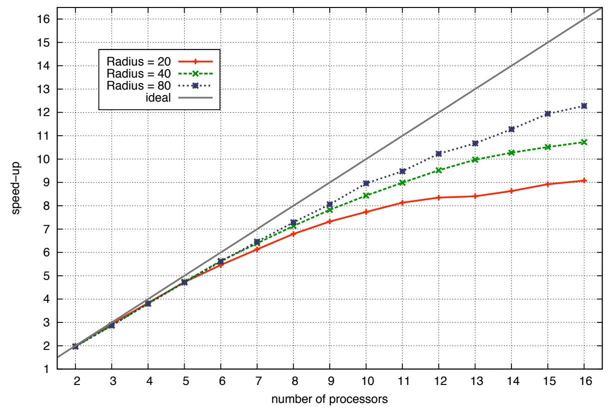

Figure 5. Speed-ups over Opteron (Case 2).

Figure 5 shows the speed-ups we have obtained over Opteron with a fixed matrix size. As it can be observed, when we have used higher radiuses we have got better speed-ups. However, as the number of processes is increased we can see a considerable speed-up decrease (specially with lower radiuses). The reason for that is the bad memory allocation control done by the operational system, since the data is not placed in such a way that performance gains can be extracted from the Opteron machine.

Table 2. Speed-ups on Opteron (radius = 40).

NP	Case study			
	1	2	3	4
4	3.81	3.86	3.87	3.98
8	7.02	7.13	7.55	7.64
12	9.08	9.52	10.34	10.65
16	10.18	10.73	12.18	12.90

The influence of the matrices sizes over Opteron can be seen in Table 2, where **NP** stands for *number of processors*. One can notice that speed-ups are higher when we use larger input matrices. Due to the fact that this machine has a low NUMA factor (from 1.2 to 1.5), even if the data is stored away from the processor which accesses it the time spent for this operation does not have a significant impact on the overall performance.

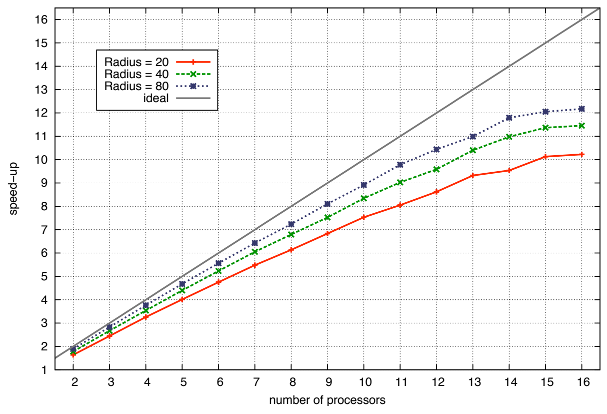

Figure 6. Speed-ups over Itanium 2 (Case 2).

The speed-ups we have obtained over Itanium 2 with a fixed matrix size can be seen in Figure 6. Similarly to Figure 5, one can see that we have better speed-ups with higher radiuses. However, when we compare Figures 5 and 6, one can notice that in Figure 6 we have got better speed-ups with lower radius (20 and 40). This is a consequence of the higher NUMA factor of this machine. The higher is the radius, the higher will be the number of remote accesses, since there is not a specific control to place data close to the processors which accesses them. Consequently, by using lower radiuses, we reduce the remote accesses impact resulting on better speed-up factors in comparison to those presented in Figure 5.

Table 3. Speed-ups on Itanium 2 (radius = 40).

NP	Case study			
	1	2	3	4
4	3.57	3.54	3.53	3.53
8	6.86	6.79	6.68	6.67
12	9.70	9.58	9.36	9.35
16	11.83	11.46	11.27	11.22

In contrast to the Opteron results, Itanium 2 showed worse speed-ups when the matrix size was increased (Table 3). Analogously to the radius variation experiments, the high NUMA factor of this machine (from 2 to 2.5) influences considerably on the overall performance, since the data locality is not controlled properly by the Linux kernel. Therefore, the high number of remote accesses reduces the speed-up factor as we increase the matrix size.

3.3. Discussion

The OpenMP ICTM parallel version has presented speed-ups around 12 for 16 processors over both machines. One can easily conclude that the lack of a better memory allocation strategy is the reason for the performance loss. As mentioned before, it is not possible to control data locality and threads placement using only OpenMP directives. A better control of data locality and threads placement can reduce the interference of non-uniform memory accesses, making it possible to improve significantly the performance gains on NUMA machines.

4. Memory Affinity Improvement

In this section, we introduce a new ICTM parallel version using memory affinity which is called NUMA-ICTM. In this solution, we have added several different memory policies provided by a library named MAI (Memory Affinity Interface). This improvement allows a better use of NUMA machines, making the categorization of large geographic regions even faster.

An alternative for the use of the MAI library would be the NUMA support present in several operating systems, such as Linux and Solaris. This support can be found at the user level (administration tools or shell commands) and at the kernel level (system calls and NUMA APIs) [6].

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

The NUMA API is an interface that defines a set of system calls to apply memory policies and processes/threads scheduling. In this solution, the programmer must change the application code to apply the policies. The main advantage of this solution is that it is possible to have a better control of memory allocation. However, developers must know low level details about the application and the architecture to manipulate directly structures such as memory pages or blocks.

4.1. MAI Interface Library

In order to provide an easy way to manage memory affinity keeping a fine control, MAI library was proposed [10]. MAI is a library developed in C that defines some high level functions to deal with memory affinity in NUMA architectures. This library allows developers to manage memory affinity for each variable/object of their applications. This characteristic makes memory management easier for developers, since they do not need to care about pointers and pages addresses like in the system call APIs for NUMA (libnuma in Linux for example [6]). Furthermore, with MAI it is possible to have a fine control over memory affinity: memory policies can be changed through application code (different policies for different phases). This feature is not allowed in user level tools like numactl in Linux.

The library implements four memory policies: *cyclic*, *cyclic_block*, *bind_all* and *bind_block*. In *cyclic* policies, the memory pages, in which the variable/object data are stored, are placed in physical memory following a round-robin strategy to store them over the memory blocks. The main difference between *cyclic* and *cyclic_block* is the amount of memory pages used to do the cyclic process. In *bind_all* and *bind_block* policies, the memory pages are placed in memory blocks specified by the developer. The main difference between *bind_all* and *bind_block* is that in the latter pages are placed in the memory blocks that have the threads/processes which will make use of them.

Besides the memory policies control, MAI also allows memory pages migration in order to optimize any incorrect memory placements.

4.2. ICTM with MAI Library

After implementing a parallel solution using OpenMP directives, we have added specific MAI functions in the code to apply memory policies and threads placement. Basically, we have modified the initialization process, in which the matrices are allocated. Two groups of functions were used to control memory and threads affinity.

Threads affinity is controlled by using the MAI bind_threads() function. With this function we can specify where each thread must be physically placed in terms of processors or CPU cores. Thus, we assure that threads migration will not occur.

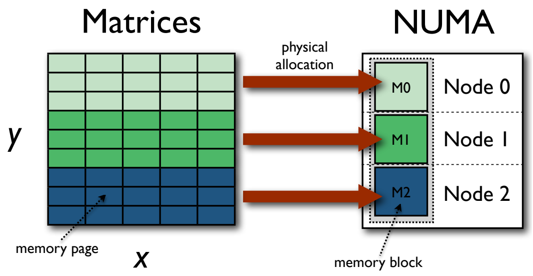

Figure 7. Bind_block policy.

Instead of using malloc() functions to allocate the matrices (as we did in OpenMP parallel solution), we have used a MAI specific function, which is called alloc_2D(). In few words, this function uses the system call mmap() to make a mapping of the physical RAM to a virtual memory. The amount of memory and the type of data to be allocated are passed through alloc_2D() parameters, similarly to malloc() function. By using alloc_2D() function, we can set a specific memory policy which will be applied to the matrices.

Figure 7 shows how the *bind_block* memory policy can be applied to ICTM matrices (the matrices cells were grouped in terms of memory pages). The memory pages, in which the matrices data are stored, are physically allocated on the NUMA memory blocks according to the work distribution done by the omp parallel for directive. Thus, each thread will access memory pages stored in the same node, reducing the number of remote accesses. On the other hand, with the *bind_all* policy, we can specify a set of memory blocks in which the matrices memory pages can be stored. However, the Linux kernel is responsible for selecting in which memory block each page will be physically allocated.

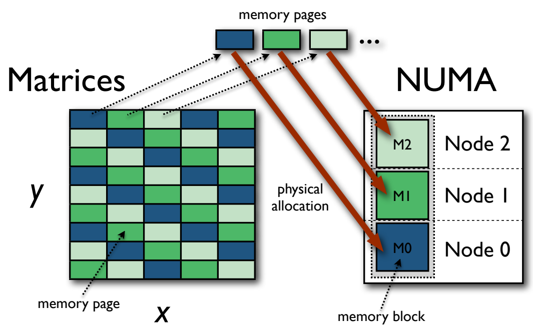

Figure 8. Cyclic policy.

When a *cyclic* policy is applied, the memory pages are physically allocated as shown in Figure 8. These memory pages are distributed among the NUMA nodes by a cyclic way: the first memory page of each matrix is physically stored on Node 0, second page on Node 1, third page on Node 2, fourth page on Node 0 and so on. A similar behavior occurs when we apply the *cylic_block* policy. However, sets of memory pages are distributed, instead of distributing page by page.

4.3. Performance Evaluation

This section shows the results obtained with NUMA-ICTM over the same platforms described in Section 3.2.1. Experiments have been done with the four memory policies implemented by MAI library. In order to compare the performance of NUMA-ICTM with different policies in both NUMA platforms, we have used the same specific case study and radius value of the Section 3.2.3: Case 2 with radius 40. Thus, we can compare the OpenMP ICTM parallel version with NUMA-ICTM.

Figure 9 shows a comparison of the four memory policies over Opteron. The *bind_all* and *bind_block* policies have presented worse speed-ups. On the other hand, one can observe that *cyclic* and *cyclic_block* have presented similar results. As mentioned before, the difference between *cyclic* and *cyclic_block* policies is the amount of memory pages used to do the cyclic distribution among memory blocks. In these experiments, the block size of the *cyclic_block* policy was a group of 10 pages. Other experiments have shown worse speed-ups as we increased the block size. Thus, it is better to use the *cyclic* policy in this machine, which distributes page by page.

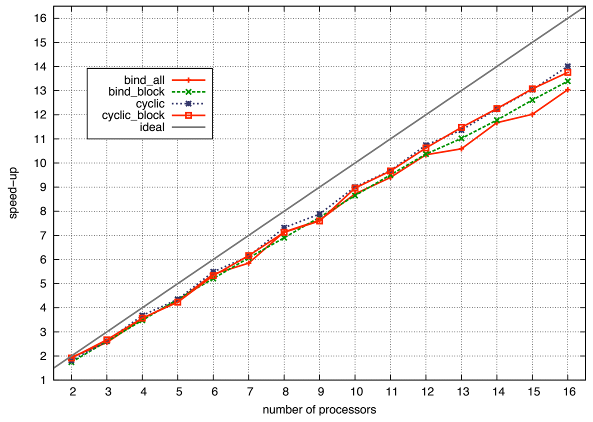

Figure 9. Speed-ups over Opteron (Case 2).

Due to the fact that Opteron has a network bandwidth problem, it is better to spread the data among NUMA memory blocks. By using this strategy, we reduce the number of simultaneous accesses on the same memory node. As a consequence of that, we can have a better performance. More detailed information about the best memory policy for Opteron machine (*cyclic*) can be seen in Table 4, in which the speed-ups of different case studies are compared.

Table 4. Cyclic policy over Opteron.

NP	Case study			
	1	2	3	4
4	2.80	3.50	3.39	3.58
8	7.71	7.32	6.74	7.10
12	11.66	10.74	10.30	10.52
16	15.34	14.01	13.67	13.50

We have done the same experiments over Itanium 2 and the results of the memory policies comparison are shown in Figure 10. One can see that these results are quite different when we compare to those obtained over Opteron (Figure 9). By allocating the rows of the matrices on the memory blocks closer to the processors which computes them, we decrease the high NUMA factor impact. As a result, *bind_block* was the best policy for this machine.

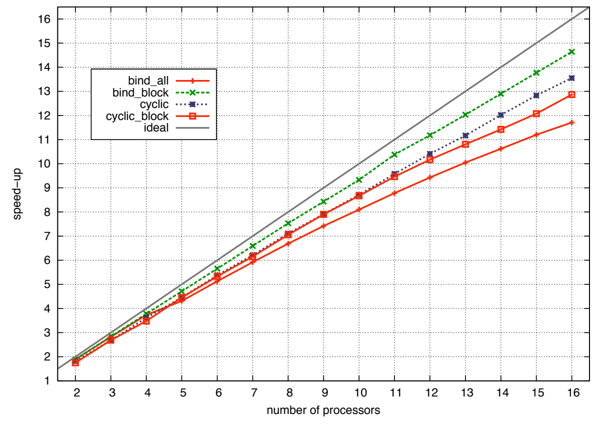

Figure 10. Speed-ups over Itanium 2 (Case 2).

Table 5 shows more information about the performance of the best memory policy over Itanium 2 (*bind_block*).

Table 5. Bind_block policy over Itanium 2.

NP	Case study			
	1	2	3	4
4	3.75	3.78	3.73	3.75
8	7.45	7.53	7.29	7.00
12	11.10	11.18	10.76	10.59
16	14.55	14.64	13.95	13.36

5. Conclusion and Perspectives

In this paper we have presented NUMA-ICTM: a parallel version of ICTM exploiting memory placement strategies for NUMA machines. First, an initial version using only OpenMP has been proposed. This solution has shown limitations, since data locality and threads placement could not be controlled. After, we have introduced the idea of memory optimization using MAI interface. MAI specific functions have been used to apply memory policies, increasing the scalability and performance of the initial version.

We have observed an overall average of 12.1% of performance gain using memory optimization in both architectures. Moreover, a higher performance gain was obtained as the number of processors was increased (average of 18% from 8 to 16 processors). The performance gain was about 23.2% if we only compare the results with 16 processors. These results were expected, since when the number of nodes increases, the number of remote accesses also grow. Thus, memory allocation policies optimizations became important.

As future works we highlight: a new version using OpenMP 3.0 and a distributed implementation to be executed in clusters of NUMA nodes.

References

[1] A. Basumallik, S.-J. Min, and R. Eigenmann. Towards OpenMP Execution on Software Distributed Shared Memory Systems. In *ISHPC '02: Proceedings of the 4th International Symposium on High Performance Computing*, pages 457–468, London, UK, 2002. Springer-Verlag.

[2] 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*, pages 48–48, Dallas, Texas, USA, 2000. IEEE Computer Society.

[3] D. Coblentz, V. Kreinovich, B. Penn, and S. Starks. Towards Reliable Sub-Division of Geological Areas: Interval Approach. In *NAFIPS '00: Proceedings of the 19th International Meeting of the North American Fuzzy Information Processing Society*, number 0-7803-6274-8, pages 368–372, Atlanta, GA, USA, 2000. IEEE Computer Society.

[4] 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*, 37(1–4):3–11, 2004.

[5] R. B. Kearfott and V. Kreinovich. *Applications of Interval Computations*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

[6] A. Kleen. A NUMA API for LINUX. Technical Report Novell-4621437, Novell, April 2005.

[7] S.-J. Min, A. Basumallik, and R. Eigenmann. Optimizing OpenMP Programs on Software Distributed Shared Memory Systems. *Int. J. Parallel Program.*, 31(3):225–249, 2003.

[8] R. E. Moore. *Methods and Applications of Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1979.

[9] T. Mu, J. Tao, M. Schulz, and S. A. Mckee. Interactive Locality Optimization on NUMA Architectures. In *SOFTVIS '03: Proceedings of the ACM 2003 Symposium on Software Visualization*, San Diego, CA, USA, 2003. ACM.

[10] C. P. Ribeiro and J.-F. Mhaut. MAI: Memory Affinity Interface. Technical Report 0359, INRIA, 2008.

[11] R. K. S. Silva, M. S. de Aguiar, C. A. F. D. Rose, and G. P. Dimuro. Extending the HPC-ICTM Geographical Categorization Model for Grid Computing. In B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *PARA*, volume 4699 of *Lecture Notes in Computer Science*, pages 850–859. Springer, 2006.

[12] 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*, pages 143–148, Washington, DC, USA, 2006. IEEE Computer Society.

[13] C. Terboven, D. an Mey, and S. Sarholz. OpenMP on Multicore Architectures. In *A Practical Programming Model for the Multi-Core Era*, pages 54–64. Springer, 2008.