A HYBRID PARALLEL VERSION OF ICTM FOR CLUSTER OF NUMA MACHINES

Mateus Raeder, Neumar Ribeiro, Dalvan Griebler, Luiz Gustavo Fernandes
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Programa de Pós-Graduação em Ciência da Computação (PPGCC)
Av. Ipiranga, 6681 - Prédio 32, 90619-900 - Porto Alegre, RS, Brazil
{mateus.raeder, neumar.ribeiro, dalvan.griebler}@acad.pucrs.br, luiz.fernandes@pucrs.br

Márcio Castro
Institut National de Recherche en Informatique et en Automatique (INRIA)
Laboratoire d' Informatique de Grenoble (LIG) - Université de Grenoble
ENSIMAG Antenne de Montbonnot, ZIRST 51, Avenue Jean Kuntzmann - 38330 - Montbonnot - France
marcio.castro@imag.fr

ABSTRACT

Recently, the emergence of new technologies enabled the creation of clusters with multiprocessor nodes. In these architectures, computing nodes are composed by more than one processor or core sharing the same memory. In some cases, processes placed on different processors or cores may have different response time when accessing memory. A cluster composed by such nodes is said to be a cluster of Non-Uniform Memory Access (NUMA) machines. In this scenario, parallelism can be better extracted using hybrid programming (MPI/OpenMP) along with memory affinity policies. In this work we propose a hybrid parallel version with memory affinity of the Interval Categorizer Tessellation Model (ICTM) for a cluster of NUMA machines. Our results show that the hybrid parallelization of ICTM allows better scalability and that we achieved speed-ups up to 40 while combining OpenMP, MPI and memory affinity.

KEYWORDS

Hybrid Programming, MPI, OpenMP, NUMA, Memory Affinity, Cluster.

1. INTRODUCTION

The High Performance Computing (HPC) scenario is nowadays composed by a wide range of different architectures and their respective communication paradigms. Clusters and Massive Parallel Processors (MPP) machines, for instance, are distributed memory based architectures and they use the message passing programming paradigm to allow communication among processes. On the other hand, Symmetric Multiprocessor (SMP) and Non-Uniform Memory Access (NUMA) machines share the same memory among processes running on different processors. Over this kind of platform, communication among processes is performed through the write and read operations over shared variables. A third alternative is the combination of these distributed and shared memory machines to create a hybrid architecture, like clusters of NUMA machines for example. In this scenario, the design and implementation of parallel programs for such architectures should be adapted to properly exploit the architecture potential. Therefore, a combination of both paradigms (message passing and shared memory) is the natural path to allow better performances through the correct use of the computational resources.

There are many libraries that implement the programming paradigms previously mentioned, however we can point out the Message Passing Interface (MPI) (Gropp, 1999) as a *de facto* standard for distributed memory programming and OpenMP (Quinn, 2004) as the most used library the for shared memory paradigm. In fact, recently, some authors (*e.g.*, Rabenseifner, 2009) claim that the combination of both (MPI and OpenMP) is emerging as a natural choice for programming hybrid HPC architectures. The most part of hybrid MPI/OpenMP codes are based on an hierarchical structure model which allows a coarse grain parallelism with MPI and a fine grain with OpenMP.

Hybrid Programming based on MPI/OpenMP combination however is not always enough to extract the best results from a hybrid architecture. In some cases, hybrid architectures may be clusters of NUMA machines. In this kind of HPC platform, each node of the cluster is a shared memory architecture in which processes have different response time when accessing the memory depending on what processor they are placed. In these machines, OpenMP alone is not the best option to extract the best performance from a NUMA node. In fact, a memory affinity programming interface is helpful in such cases since it may offer primitives to control processes placement in such way they can access memory in the fastest possible way.

In this work, we intend to take advantage of the Hybrid Programming flexibility along with memory affinity policies to create a new parallel version for the Interval Categorizer Tessellation Model (ICTM) (Aguiar, 2004) for clusters of NUMA machines: the **Hybrid ICTM**. ICTM is a multi layered model based on the concept of tessellations employed to categorize geographical areas from satellite images. The model is capable to identify in separated layers different characteristics such as: vegetation, topography, land use, etc. ICTM is a helpful tool to analyze and comprehend the use of a given region. For its importance, parallel versions of ICTM have been developed in the past years (Silva, 2006 and Castro, 2009) in order to improve its usage allowing the computation of larger images in a faster time. An ICTM hybrid parallel version is the next step in this direction, since clusters of NUMA machines are becoming cheaper and wide spread.

The remaining of this work is organized as follows: in Section 2, we introduce some related works; in Section 3 we briefly discuss NUMA architectures and the programming tools we have used; our proposed version for ICTM over hybrid architectures is presented in Section 4; performance experiments are described and their results are shown in Section 5 and finally Section 6 brings some conclusions.

2. RELATED WORK

In recent years, many research groups used a hybrid parallel model based on the combination of MPI/OpenMP to improve the resources exploitation of hybrid architectures.

One decade ago, research works were more exploratory. In the work of (Cappello, 2000), a comparison between a hybrid version and a pure MPI version of the NAS benchmark was carried out. Authors have observed that performance relays on aspects such as memory access patterns and hardware configuration. In (Henty, 2000), the authors describe a case study (parallel versions for the discrete element modeling) in which they have developed pure MPI, pure OpenMP and hybrid solutions. Their results show that both pure MPI and pure OpenMP versions presented better performance than the hybrid code. In another work (Smith and Bull, 2000), authors claim that the hybrid programming model can outperform pure MPI implementations for some applications, but it is not optimal for all scenarios. In (Jost, 2003), authors also used the NAS benchmark and concluded that the hybrid model fits better architectures based on slower interconnection networks. In 2005, good performances were achieved applying hybrid programming for applications running over clusters of SMP machines (*e.g.*, Ashworth, 2005).

More recently, authors successfully focused on adapting applications with huge computational demand using a hybrid programming model to exploit more powerful architectures. They also proposed mechanisms to make it easier to program using the hybrid model. The authors of (Lusk and Chan, 2008), studied how MPI processes interact with OpenMP threads and proposed a tool to investigate these interactions during the execution of an application. In (Chorley, 2009), authors used a molecular dynamics code to compare a hybrid implementation to a pure MPI version of the same code. In their conclusions, they inform that for some parts of the code the hybrid version achieved better results over a cluster of multicore machines. Finally, the work of (Rabenseifner, 2009) introduces an overview about hybrid programming in current days, describing which are the main hybrid parallel programming practices that result in better performances.

3. BACKGROUND

Traditional UMA (Uniform Memory Access) architectures such as SMPs present a single memory controller, which is shared by all processors. This single memory connection often becomes a bottleneck when many processors access 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.

In contrast, 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 (Lameter, 2011). These machines have multiple memory levels that are seen by the developers as a single memory. They combine the efficiency and scalability of MPP architectures with the programming facility of SMPs. 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) (Bolosky, 1991).

The effects of such "asymmetry" on NUMA architectures can be reduced by optimizing memory affinity (Ribeiro, 2009). Memory affinity is assured when a compromise between thread and data is achieved through the use of memory policies by either reducing the number of remote accesses or the memory contention (Ribeiro, 2010). In this context, first-touch is the default policy in the Linux operating system to manage memory affinity. This policy places data on the node that first accesses it. However, it is possible to use different strategies do place data on NUMA nodes.

Although the NUMA API (Kleen, 2011) is the most common library available on Linux to deal with memory policies, there are other higher level solutions such as Minas framework (Ribeiro, 2010). Beyond the architecture abstraction, this framework also provides several memory policies allowing better memory access control of parallel applications for NUMA architectures.

In this paper we take into account all these factors while developing a hybrid parallel version of ICTM for clusters of NUMA machines. The parallelization is performed in two different levels (inter- and intra-node). The intra-node parallelization is strongly based on the work presented in (Castro, 2009). In that work, ICTM was parallelized with OpenMP and tuned with memory affinity policies to obtain better performance gains on NUMA architectures. The main contribution of this work is the inclusion of an inter-node parallelization which combines OpenMP with MPI to parallelize the application across different cluster nodes.

4. HYBRID-PARALLEL ICTM

ICTM is a multi-layered tessellation model for the categorization of geographic regions considering several different characteristics such as relief, vegetation and climate (Aguiar, 2004). 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 subdivisions of the regions considering the simultaneous occurrence of all characteristics. This allows interesting analyses about their mutual dependencies.

The input data is extracted from satellite images, in which the information is given in certain points referenced by their latitude and longitude coordinates. The geographic region is represented by a 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.

The categorization process is performed per layer. This means that all layers must be categorized individually before the final projection computation. In each layer the categorization is composed by several phases, where each phase uses the results obtained from the previous one (Figure 1).

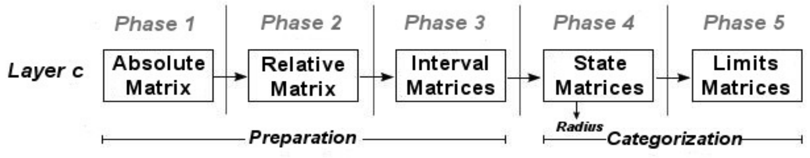

Figure 1. Categorization process phases

Categorizing extremely large regions has a high computational cost. This cost is basically related to two parameters: the size of the matrices and the number of neighbors that are analyzed during the categorization process in each layer (radius). The higher are the size of the matrices and the number of neighbors the higher will be the computation cost.

The hybrid parallel version of ICTM proposed in this work uses the message passing paradigm to distribute the tasks among the cluster nodes (inter-node parallelization) and the shared memory paradigm within the NUMA nodes (intra-node parallelization).

The inter-node parallelization uses MPI and follows a master/slave approach. In the first phase, the master process divides the input matrix to be computed into "n" blocks, where "n" is the number of the cluster nodes (Figure 2). Then, it sends to each slave the information needed to compute its own block. However, the computation of each cell requires information from its neighbors. As explained before, the number of neighbors is defined by the application parameter called "radius". Because of that, the block sent to each slave has additional columns on both sides (left and right), considering the size of the radius. Thus, slaves do not need to exchange messages to gather information about neighbors of border cells, improving the performance of the parallel solution. After finishing the computation of the final phase, the master process is responsible for combining all computed blocks excluding the redundant cells.

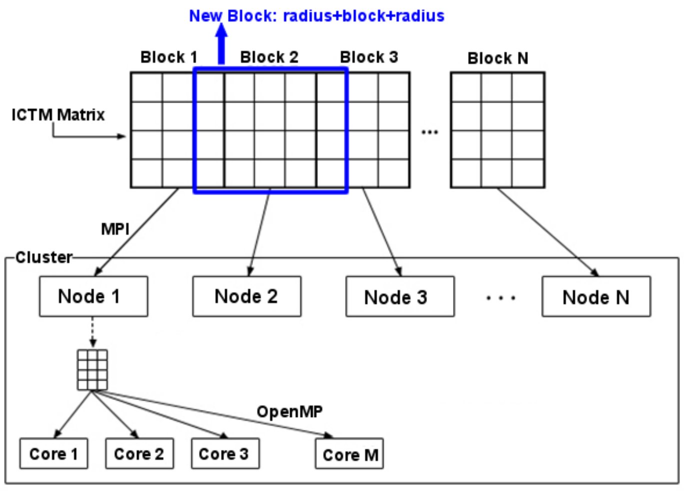

Figure 2. Tasks division among the process in the hybrid architecture

The intra-node parallelization is done by each slave process in the cluster node. Since we consider nodes composed by multicore machines, we use OpenMP to parallelize the computation of each block. 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 a fork-join model, which can be easily used with the ICTM sequential code to compute the matrices blocks.

In a simplified way, computations are performed in a similar way (two "nested for loops"). 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 block, as follows:

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

We believe this is a simple and elegant solution, since we only make few changes in the sequential source code. However, OpenMP directives do not allow controlling memory allocation among the NUMA nodes. Thus, in the intra-node parallelization we discuss two solutions. First, we use only OpenMP to decompose tasks among threads (Section 5.1) and then we tune the OpenMP parallelization considering the NUMA characteristics of the cluster nodes (Section 5.2). Specifically, we use MAi (Memory Affinity interface)

(Ribeiro, 2010) to improve the memory affinity. For more details about the OpenMP parallelization and the memory affinity strategies applied to ICTM please refer to (Castro, 2009).

5. OBTAINED RESULTS

In this section we present the results of the Parallel-Hybrid ICTM. The tasks division is performed as follows: each process is responsible for processing a block of the ICTM matrix. For that, the work is divided among the nodes considering only the columns division, i.e., if the input matrix has dimensions 15,000x15,000, for example, and the application will be executed in 6 nodes of a cluster, each one will receive a block with dimensions 15,000x2,500 to process.

The work described in (Castro, 2009) used different sizes of matrices (between 4,800x4,800 and 13,300x13,300) and radius considering 20, 40 and 80 neighbors. As we considered a hybrid solution that is peculiarly more scalable, we worked with matrices with dimensions 10,000x10,000 and 15,000x15,000, with the same radius values (20, 40 and 80), since they are appropriate values for the tests. Tests with larger matrices are not shown because they use the swap memory, and it is not interesting for the performance analysis.

The results were obtained from the average of 10 executions, where the standard deviation was always between 0 and 3 seconds. The experiments were performed with exclusive access to the nodes of a cluster composed of 10 Dell PowerEdge R610 machines, each one with 2 Intel Xeon Quad Core E5520 2.27GHZ Hyper-Threading processors, resulting a total of 16 cores per node (8 physical and 8 logical), 16GB of RAM and 150 GB of disk storage. The nodes are connected by two Gigabit Ethernet switched networks, one for communication among the nodes and one for management. The Intel Xeon Processor E5520 has 8M of cache, clock 2.27GHz and 5.86 GT/s Intel QPI. The mpich (implementation of MPI) and OpenMP versions used were 1.2.7p1 and 1.4.2, respectively.

5.1 Parallel-Hybrid ICTM

In this section, the obtained results after the execution of the Parallel-Hybrid ICTM only with MPI and OpenMP are presented. Table 1 shows the results (in seconds) using 6 nodes of the cluster. It is important to notice that the MAi was configured to use only one core of each processor. The first part of Table 1 represents the times obtained for a matrix of dimensions 10,000x10,000, and the second part (with gray background) represents the times for a matrix of dimensions 15,000x15,000. The line in bold is the ICTM sequential time.

Table 1. Hybrid-Parallel ICTM obtained times

	Matrix 10,000x10,000			Matrix 15,000x15,000		
Cores	Radius 20	Radius 40	Radius 80	Radius 20	Radius 40	Radius 80
1	**75.12**	**129.98**	**248.60**	**173.98**	**301.22**	**564.97**
6	15.11	26.23	49.95	34.68	60.38	112.89
12	7.75	13.32	25.17	17.66	30.53	56.92
24	4.05	6.84	12.79	9.05	15.51	28.70
48	3.67	5.73	9.56	7.30	11.53	18.79
96	4.24	5.58	8.40	7.78	10.77	17.18

As seen in Table 1, the hybrid version execution times are smaller than the sequential ones. For a matrix of dimensions 10,000x10,000, for example, with only 6 cores (i.e., one core per node) the hybrid version presented a decrease of approximately 80% in the execution time for all values of radius. When more cores are added, the execution times continue to decrease until 48 cores, where it can be seen a reduction around 95% for all radii. However, with the execution using 96 cores it is clear that there is not a significant gain compared to results with 48 and 24 cores, but we can still note a great gain in relation to the sequential version. With a larger matrix (15,000x15,000), the behavior of the hybrid version was similar, decreasing in 97% the execution times when the use of 96 cores and radius 80.

Figure 3 presents the speed-up graphics for the Parallel-Hybrid ICTM. The curves show that when the application is performed over 6, 12 and 24 cores, the speed-ups continue close to the ideal. When more cores are introduced, the distance of the ideal speed-up increases. Nevertheless, using 96 cores and radius 80, for example, the hybrid version with matrix of dimensions 10,000x10,000 and 15,000x15,000 runs approximately 30 and 33 times faster than the sequential, respectively.

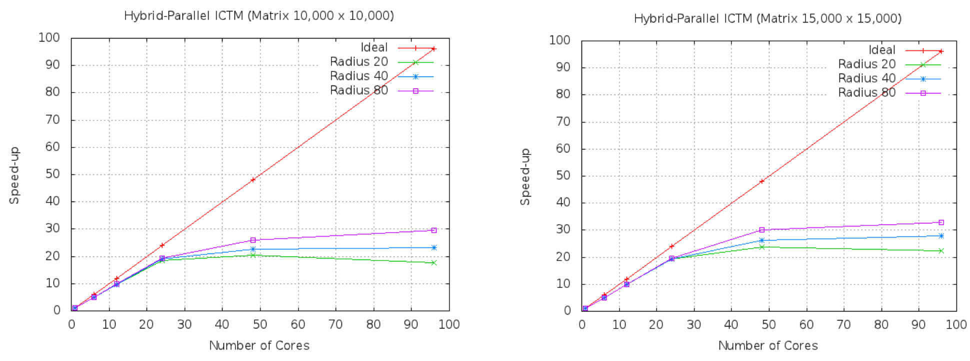

Figure 3. Hybrid-Parallel ICTM speed-ups

5.2 Parallel-Hybrid ICTM with Memory Affinity

In this section we present the results obtained after execution of the Parallel-Hybrid ICTM with Memory Affinity. The difference to the previous implementation is that this version uses the MAi library to explore memory affinity in the NUMA architecture. Table 2 shows the execution times for this version, for matrices of dimensions 10,000x10,000 and 15,000x15,000 (gray background). The highlighted line refers to the sequential time of the ICTM.

Table 2. Hybrid-Parallel ICTM with Memory Affinity obtained times

	Matrix 10,000x10,000			Matrix 15,000x15,000		
Cores	Radius 20	Radius 40	Radius 80	Radius 20	Radius 40	Raio 80
1	**75.12**	**129.98**	**248.6**	**173.98**	**301.22**	**564.97**
6	15.22	26.24	50.20	34.06	59.16	113.49
12	7.66	13.19	25.16	17.15	29.73	56.84
24	3.82	6.61	12.59	8.55	14.89	28.45
48	1.93	3.32	6.31	4.30	7.48	14.29
96	1.83	3.21	5.95	4.10	7.32	13.79

The hybrid version with memory affinity also achieved much better results than those observed in the sequential version. With radius 80 for both dimensions of matrices, the execution time with 96 cores represents an improvement of approximately 97%. Unlike the previous version, it is clear that the execution times decrease when more cores are added, including the execution over 96 cores. However, the differences between the times are reducing more and more, indicating that the use of more than 96 cores will not bring greater gains.

When these results are compared with the results of the previous version that does not use memory affinity (Table 1), it is noticeable an interesting reduction of the execution times when more cores are used. With a matrix of dimensions 15,000x15,000, for example, the version without memory affinity takes 4.24s to run over 96 cores, while the version with memory affinity takes only 1.83s.

Figure 4 presents the speed-ups obtained for this version. For the execution with 6, 12, 24 and 48 cores the speed-up curve remains close to the ideal, reaching a speed-up of almost 40 for 48 cores. With the use of 96 cores, the gain over the 48 cores is small, but still very large when compared to the sequential time, being the hybrid version with memory affinity about 40 times faster. These results (as well as the previous version) clearly illustrate that the use of logical cores (obtained with hyper-threading) results in a bottleneck in the memory access. However, the execution times and speed-ups obtained even with the use of hyper-threading feature are highly satisfactory.

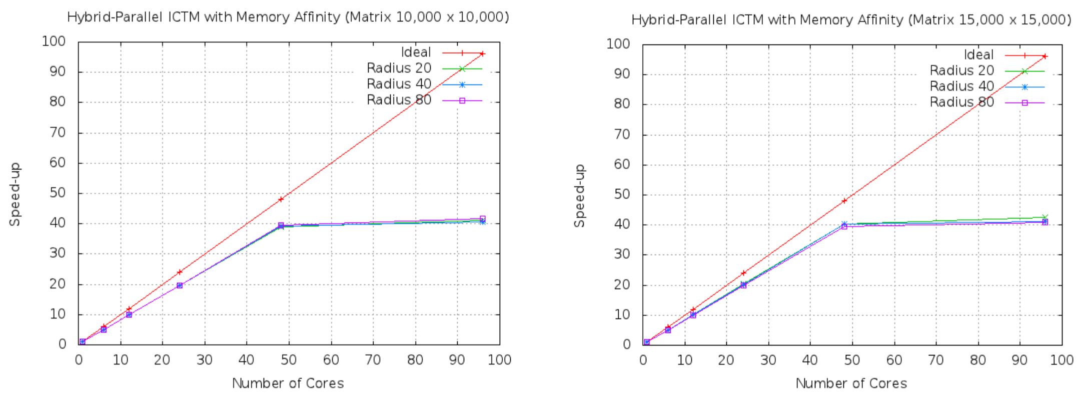

Figure 4. Hybrid-Parallel ICTM with Memory Affinity speed-ups

Finally, Figure 5 shows the comparison of the hybrid versions according to their efficiency, related to the execution with radius 80 as an example. It can be seen that both versions have good values of efficiency (80%) using up to 24 cores. The memory affinity version continues with the efficiency of approximately 80% with 48 cores, while the values of the other version start to decrease. Overall, the version that uses resources to improve the memory access presents better efficiency than that which uses only MPI and OpenMP. The efficiency found in both versions (especially those with memory affinity) shows that the implemented solution is highly scalable, and more cores can be used, obtaining values still much better than those found in the sequential version.

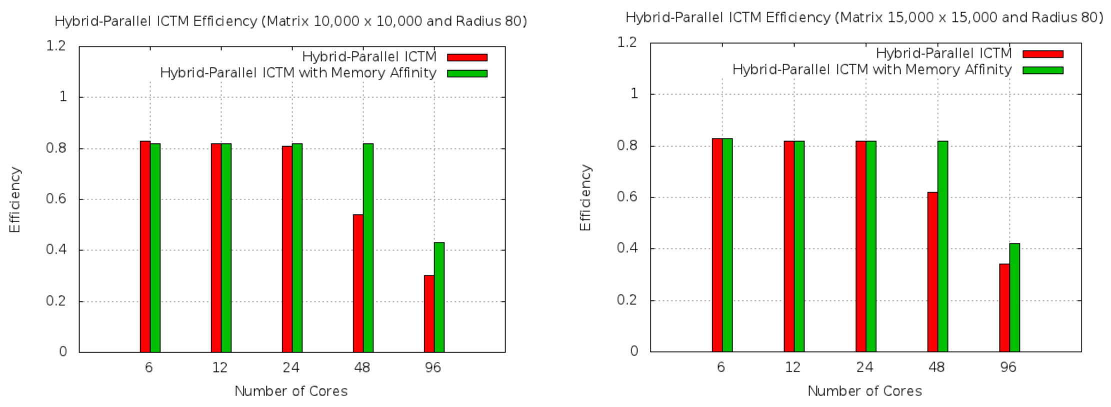

Figure 5. Comparison of the Hybrid-Parallel ICTM versions efficiency

6. CONCLUSION

This study aimed at implementing a parallel hybrid version for the ICTM application in NUMA architectures. Two hybrid versions were developed, one only with MPI and OpenMP, and another that uses the MAi library to explore memory affinity. Results show that the greater the number of cores used the greater the gains of the hybrid version that uses memory affinity, improving the execution time up to 97% when compared with the sequential time. Another important aspect is that the version implemented in

(Castro, 2009) has a limitation on the number of cores, according to the NUMA machine being used. In the hybrid version presented in this paper we used 96 cores with a very good performance gain.

The efficiency of the versions is in most cases greater than 80%, which shows that the developed version is highly scalable. Thus, larger matrices can be used and more cores can perform the computation. As an example, the hybrid version with memory affinity was performed with 6 nodes and a matrix 40,000x40,000. Despite using the swap area, which reduces the application performance, the obtained times were 100.3s with 24 cores, 50.99s with 48 cores and 51.56s with 96 cores.

REFERENCES

Gropp W., Lusk E. and Skjellum A., 1999. *Using MPI*. The MIT Press, Cambridge, Massachusetts.

Quinn J. M., 2004. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, NY.

Rabenseifner R., Hager G. and Jost G., 2009. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. *In Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing(IEEE Computer Society)*. Washington, DC, USA, pp. 427-436.

Corrêa, M., Zorzo A. and Scheer R., 2006. Operating System Multilevel Load Balancing. *In Proceedings of the 2006 ACM symposium on Applied computing*. New York, NY, USA, pp. 1467-1471.

Aguiar, M. S., 2004. The Multi-layered Interval Categorizer Tesselation-based Model. *In Proceedings of the 6th Brazilian Symposium on Geoinformatics*. Campos do Jordão, São Paulo, Brazil, pp. 437-454.

Silva R. K. S., Aguiar M. S., De Rose C. A. F. and Dimuro G. P., 2006. Extending the HPC-ICTM Geographical Categorization Model for Grid Computing. *In: Workshop on State-of-the-art in Scientific Computing (PARA)*. Umeá, Sweden, pp. 850-859.

Castro M., Fernandes L. G., Ribeiro C. P., Méhaut J.-F. and Aguiar M. S., 2009. NUMA-ICTM: A Parallel Version of ICTM Exploiting Memory Placement Strategies for NUMA Machines. *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. Rome, Italy, pp. 1-8.

Cappello F. and Etiemble D., 2000. MPI Versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. *In Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM) (Supercomputing '00)*. IEEE Computer Society, Washington, DC, USA.

Henty, D. S., 2000. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. *In Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM) (Supercomputing '00)*. Washington, DC, USA.

Smith L. and Bull M., 2000. Development of Mixed Mode MPI/OpenMP Applications. *Sci. Program*. Vol. 9, No 2,3, pp. 83-98.

Lusk E. and Chan A., 2008. Early Experiments with the OpenMP/MPI Hybrid Programming Model. *In Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*. West Lafayette, IN, USA, pp. 36-47.

Jost G., Jin H., Mey An D. and Hatay F. F., 2003. *Comparing the OpenMP, MPI, and Hybrid Programming Paradigm on an SMP Cluster*. NASA Ames Research Center, Aachen, Germany.

Ashworth M., Bush J. I., Guest F. M., Sunderland A. G., Booth S., Hein J., Smith L., Stratford K. and Curioni A., 2005. HPCx: Towards Capability Computing: Research Articles. *Concurr. Comput.: Pract. Exper*. Vol. 17, No. 10, pp. 1329-1361.

Chorley J. M., Walker W. D. and Guest F. M., 2009. Hybrid Message-Passing and Shared-Memory Programming in a Molecular Dynamics Application On Multicore Clusters. *Int. J. High Perform. Comput. Appl*. Vol. 23, No. 3, pp. 196-211.

Lameter C., 2011. Local and Remote Memory: Memory in Linux/NUMA System. Extracted from <http://ftp.kernel.org/pub/linux/kernel/people/christoph/ols2006/ols2006.pdf>. Last access: September 5th, 2011.

Bolosky, J. W., Scott L. M., Fitzgerald, R. P., Fowler R. J. and Cox A. L., 1991. NUMA Policies and their Relation to Memory Architecture. *SIGPLAN Not*. Vol. 26, No. 4, pp. 212-221.

Ribeiro P. C., Méhaut J.-F., Carissimi A., Castro M., and Fernandes L. G., 2009. Memory Affinity for Hierarchical Shared Memory Multiprocessors. *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Sao Paulo, Brazil, pp. 59-66.

Ribeiro P. C., Castro M., Méhaut J.-F., and Carissimi A., 2010. Improving Memory Affinity of Geophysics Applications on NUMA Platforms Using Minas. *International Meeting on High Performance Computing for Computation Science (VECPAR)*. Berkeley, USA, pp. 272-292.

Kleen A., 2011. A NUMA API for LINUX. Extracted from <http://www.halobates.de/numaapi3.pdf>, Last access: September 5th, 2011.