Effective Use of Multicore Clusters in Parallel Cellular Automata

A. Marcela Printista[1 2] and Fernando Saez[1]

[1] Laboratorio de Investigación y Desarrollo en Inteligencia Computacional
[2] Universidad Nacional de San Luis
[3] CONICET CCT-San Luis-Argentina
{mprinti, bfsaez}@unsl.edu.ar

Abstract. Cellular automata provide an abstract model of parallel computation that can be effectively used for modeling and simulation of complex phenomena and systems. We start from a template designed to facilitate faster D−dimensional cellular automata application development. The key for the use of the template is to achieve an efficient implementation, irrespective of the application specific details. In the parallel implementation on a cluster was important to consider issues such as task and data decomposition. With multicore clusters, new problems have emerged. The increasing numbers of cores per node, caches and shared memory inside the nodes, has led to the formation of a new hierarchy of access to processors. In this work we discuss and evaluate strategies that will be important in optimizing prototype to run on multicore cluster. The underlying idea in our proposal is the establishment of a relation among parallel processes based on the communication topology that arises in the implementation of task division functions. We propose that this relation can efficiently map on the multicore cluster topology. We introduce a new mapping strategy that can obtain benefit in the performance by adapting its communication pattern to the hardware affinities among processes allocated in different cores. We apply our approach to a two-dimensional application achieving sensible execution time reduction.

Keywords: Parallel Programming, Cellular Automata, Multicore Nodes, Mapping Strategy

1 Introduction

Multicore processors have emerged and are currently the mainstream of general purpose computing. Quad-core processors are currently commonplace and core count by chip is expected to increase drastically in the forthcoming years. In HPC, these processors have been used as building blocks for cluster of multiple sizes, by grouping together a variable number of nodes (each containing a few multicore processors) through a commodity interconnection fabric such as Gigabit Ethernet. A real challenge for parallel applications is to exploit such architecture at their potential. In order to achieve the best performance, many new factors must be considered and studied.

MPI programming model implicitly assumes the message passing as interprocess communication mechanism, so any existing MPI code can be employed without changes

for running in a multicore cluster. The MPI conceptual model considers a set of processes which communicate with each other, regardless of whether they will be mapped to multiple cores sharing memory within a single node.

But, the technology in study involves to consider the different kind of communications among processes, depending on whether they are running on different cores within the same node or different nodes. Chai et al. [3] presented communication schemes on multicore clusters, where intra-chip, intra-node and inter-node are described. The speed of communication among cores in a multicore processor chip (intra-chip) varies with core selection, since some cores in a processor chip share certain levels of cache and others do not. Consequently, intra-chip interprocess communication can be faster if the processes are running in cores with shared caches than otherwise. This asymmetry in communication speed may be worse among cores on distinct processor chips in a cluster node (intra-node) and is certainly worst if communicating cores belong to distinct nodes of a cluster (inter-node).

As an alternative to the pure MPI model, Rabenseifner et al. [8] presented the available programming models on hybrid/hierarchical parallel platform. The authors outline that to seem natural to employ a hybrid programming model which uses OpenMP [1] for parallelization inside the node and MPI for message passing between nodes. It can expect hybrid models to have positive effects on parallel performance. However, mismatch problems arise because the main issue with getting good performance on hybrid architectures is that none of the common programming models fits optimally to the hierarchical hardware.

MPI library gives the programmer control over the decomposition task and the management of communication/synchronization among the parallel processes. Although MPI does not include explicit mapping primitives, most of its implementations have a static programming style. In this case, MPI support can avoid the mapping and scheduling problems, however, the programmer's task becomes more complex. Its unstructured programming model based on explicit, individual communications among processors is notoriously complicated and error-prone.

To reduce the software complexity without lowering the performance, an approach exists to restrict the form in which the parallel computation can be expressed. This can be done at different abstraction levels. The model provides programming constructs: skeletons, that correspond directly to frequent parallel patterns. The programmer expresses parallelism using a set of basic predefined forms with solution to the mapping and restructuring problems.

Simulations based on Cellular automata are ideally suited for parallel computing and consequently, researchers from diverse fields require support to design and implement parallel cellular algorithms that are portable, efficient, and expressive. Following this approach, Saez et al. [5] implemented a versatile cellular automata skeleton and an environment for its use. The skeleton is written in `C` and MPI and is accessed through a call to the constructor `CA_Call` and its parameters list allows substantial flexibility, which will bring benefits in different application domains. The skeleton enables us to write CA algorithms in an easy way, hiding parallel programming difficulties while supporting high performance.

The goal of this paper is to describe the optimization techniques applied to a CA skeleton that will be able to take advantage of a multicore environment. We will show a performance improvement not due to modifications of the MPI implementation itself but rather due to a relevant process placement.

The rest of the paper is organized as follows. Section 2 introduces a background about cellular automata and the section 3 briefly describes some important aspect of the parallel implementation of CA skeleton. In this section some experimental results of the CA Implementation on a Cluster are showed. Section 4 describes the multicore environments and focuses in the process placement and mapping problems. Section 5 presents the comparative analysis of strategies and the conclusions are given. Finally, the section 6 presents our current research activities.

2 Cellular Automata Background

Cellular automata are simple mathematical idealizations of natural systems. They consist of a $D-$dimensional lattice of cells of uniform size connected with a particular geometry, and where each cell can be in one of a finite number of states. The values of the cells evolve in discrete time steps according to deterministic rules that specify the value of each cell in terms of the values of neighboring cells and previous values. Formally, a cellular automaton is defined as a $4-$tuple (L_D, S, V, f) where L_D is a $D-$dimensional lattice partitioned into cells, S is a finite set of states ($|S| = v$) , V is a finite set of neighborhood indexes, and $\delta : S^v \rightarrow S$ is a transition function.

Below we summarize the most important characteristics that define the behavior of CA:

Initial State: The initial configuration determines the dimensions of the lattice, the geometry of the lattice, and the state of each cell at the initial stage.

State: The basic element of CA is the cell. Each cell in the regular spatial lattice, can take any of a finite number of discrete state values. In the simplest case, each cell can have the value 0 or 1. In the more complex case, the cells take more values. It is even possible to have each cell with a complex structure with multiples values.

Neighborhood: For each cell of the automaton, there are a set of cells called neighborhood (usually including the cell itself). A characteristic of CA is that all cells have the same neighborhood structure, even the cells at the boundary of a lattice have neighboring cells that could be outside the domain. Traditionally, *border cells* are assumed to be connected to the cells on the opposite boundary (that is, for one dimension, the right most cell is the neighbor of the left most one and vice versa).

Transition function: The set of rules that define how the state of each cell changes on the basis of its current state and the states of its neighbor cells. In a standard CA, all cells are updated synchronously.

3 High Performance Simulation for CA Models

From a computational point of view, CA are basically a computer algorithm that is discrete in space and time and operates on a lattice of cells. The Fig. 1 shows a simple algorithm that solves a two-dimensional generic cellular automaton. The algorithm

takes as input a two-dimensional lattice of $(N \times N)$ and initializes the structure with some initial configuration. The simulation involves an iterative relaxation process. This process is represented in the algorithm with a iteration of $steps$ steps. In each time step t, the algorithm updates each cell in the lattice. The next state of an element $s^{t+1}(i, j)$ is a function of its current state and the values of its neighbors. The relaxation process ends after $steps$ iterations.

```
1  CelAuto(Lattice,steps)
2  init(Lattice)
3  for t = 1 to steps
4        for i = 1 to N
5             for j = 1 to N
6                  nextState(Lattice,i,j)
```

Fig. 1. Sequential CA approach

Cellular automata parallel systems allow to user exploit the inherent parallelism of cellular automata to support the efficient simulation of complex systems that can be modeled by a very large number of simple elements with local interaction only. In fact, it is possible to exploit the data parallelism intrinsic to the CA programming model coming from the possibility to execute the transition function on different sublattices due to the local nature of cell interactions. The cellular space of the automaton is represented by an array of D dimensions (D-lattice), which contains N^D objects called cells. Inside of a cluster based on distributed memory system, the parallel execution using P processors (denoted $p_0, p_1, ...p_{P-1}$) is performed by applying the transition function simultaneously to P sublattices in a $SPMD$ way.

The skeleton has a component that implements the CA parallel paradigm and frees the user to consider details involved in high performance computing. It also has a component specific to the CA application that implements the transition function. In this case, the rules are described by a function built by the user.

As a first task of implementation it is necessary to find a division criterion to provide P sublattices of the automaton. The underlying idea for the implementation of lattice division functions is the establishment of a relation among P processes. The structure of divisions produced by the proposed scheme and the partnership relation established among processes give place to communication patterns that are topologically similar to a $Mesh$. This partnership is responsible of the assimilation the communication topology of a CA.

If $\sqrt[\mathcal{D}]{P}$ is a natural number and N is multiple of P, then a D-lattice can be divided in P sublattices given place to a $\mathcal{D}$-dimensional mesh where P determines the number of divisions produced on the lattice and $\mathcal{D}$ defines the dimensionality of the communication Mesh.

Applying different decompositions for the same D-dimensional lattice, different $\mathcal{D}$-dimensional meshes ($1 \leq \mathcal{D} \leq D$) may be possible, where the dimensionality of the resulting mesh will be: $\underbrace{\sqrt[\mathcal{D}]{P}^{(D-\mathcal{D}+1)}}_{d_1} \times \underbrace{\sqrt[\mathcal{D}]{P}}_{d_2} \times ... \times \underbrace{\sqrt[\mathcal{D}]{P}}_{d_\mathcal{D}}$ (for example, for a CA

involving a three-dimensional lattice, one-, two-, and three-dimensional decompositions are possible).

Once the data are divided, the skeleton implementation assigns each sublattice to a processor and lets the nodes update them simultaneously. We are developing MPI application, so, the number of ranks is the same as the number of nodes. A static mapping between the MPI processes and the processors is computed before launching the CA application. This placement will not need to be modified during the application execution.

Independently of the topology, each processor will be responsible of a sublattice, which means the evolution of $\frac{N^D}{P}$ cells. No matter what the simulation problem is attacked, a cell changes its current value through a set of rules that define its next state depending on its current value and the value of its neighboring cells.

If a cell and its neighbors are in the same node, the update is easy. On the other hand, when nodes want to update the border cells, they must request the values of the neighboring cells on other nodes. The solution to this problem is to let two neighboring lattices overlap by one row or column vector. After a node updates its interior elements, it exchanges a pair of vectors with each of the adjacent nodes. The overlapping vectors are kept in the boundary elements of the sublattices. If a neighboring node does not exist, a local boundary vector holds the corresponding boundary elements of the entire lattice.

The implementation uses asynchronous communications to allow the simultaneous advance in the exchange of the borders between neighboring processes in the $\mathcal{D}$ Mesh. Once the communication phase is completed, each process can perform the computations on their cells. The iterative process is repeated as necessary.

3.1 Experimental Results of the CA Implementation on a Cluster

In this section, we present some results obtained by using the skeleton prototype previously described, which does not consider multicore facilities. As a starting point, we propose to evaluate the performance of the described implementation, and then compare it with the approaches proposed in the next sections.

The cluster used for the experiments was a 32 Node IBM 3550, which is equipped with two Dual-Core Intel(R) Xeon(R) 3.00GHz per node. Each node has 4MB L2 (2x2). The nodes are interconnected by a Gigabit Switch.

The `CA_Call` prototype was applied to resolve the numerical solution of Laplace's equation by lattice relaxation, which is representative of the class of two-dimensional CA models we study. The problem considers Von Neumann neighborhood, which comprises the four cells (North/South/ East/West) orthogonally surrounding a central cell on a two-dimensional lattice. The processes were distributed in a round robin fashion among the 32 quad-core nodes of the cluster.

For both partitioning schemes, the Fig. 2 shows the experiments called `n-1c` where Mesh 2x2 and Mesh 4 refer to 4 nodes and Mesh 4x4 and Mesh 16 refer to 16 nodes. All nodes are usable by the MPI processes with the restriction that only a single MPI process runs on a given node. With this scheme, the operating system chooses on which core of the node a process is executed.

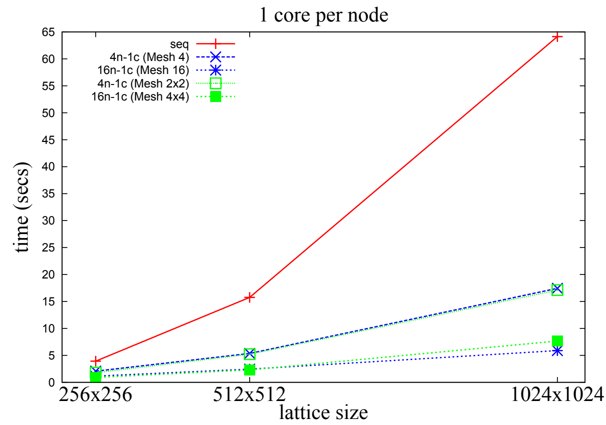

Fig. 2. $1\mathcal{D}$ and $2\mathcal{D}$ Partitioning. The benchmark places only one MPI task per node to avoid intra-node messaging

In the experiment `4n-1c`, we observed that independent of the value of $\mathcal{D}$, for the same lattice size, the analyzed schemes agree sublattice size and number of cells that must be exchanged. While in the experiment `16n-1c`, for a given lattice size, both schemes agree on the number of cells to be computed, but in the case of $\mathcal{D} = 1$ is double the number of cells required to be exchanged with its neighbors to allow parallel processing of sublattices. The Table 1 shows more precisely the situation. In the column communication size, the first term corresponds to the number of neighbors for each cell, the second term is due to the "send to & receive from" operations and the third term is the size of the data to be transmitted. The first two terms are the number of messages sent or received required for the processing of a sublattice.

Experiment	$\mathcal{D}$	Number of cells	Communication Size
4n-1c	1 (Mesh 4)	$(N/4) * N$	$2 * (2 * (N * size(cell)))$
	2 (Mesh 2x2)	$(N/2) * (N/2)$	$4 * (2 * (N/2 * size(cell)))$
16n-1c	1 (Mesh 16)	$(N/16) * (N)$	$2 * (2 * (N * size(cell)))$
	2 Mesh (4x4)	$(N/4) * (N/4)$	$4 * (2 * (N/4 * size(cell)))$

Table 1. Impact of Sublattice Size and Communication Size in $1\mathcal{D}$ and $2\mathcal{D}$ meshes

In addition to overall execution time, we measured the computation times for each lattice size. The Table 2 shows that using 4 processors (`4n-1c`), the impact of computation time is very high, obtaining, for example, in the case of an automaton of one million of cells a ratio of more than 90% of time involved in computation. As the lattice size increases, a speedup close to ideal can be visualized in the Table 3.

Lattice Size	$\mathcal{D}$	4n-1c		16n-1c	
		Total Time	Computation	Total Time	Computation
256x256	1	2,06	1,18 (57,33%)	1,16	0,36 (31,20%)
	2	1,78	1,21 (68,07%)	0,91	0,36 (39,94%)
512x512	1	5,36	4,37 (81,62%)	2,44	1,30 (51,60%)
	2	5,22	4,38 (83,95%)	2,31	1,21 (52,32%)
1024x1024	1	17,45	15,99 (91,68%)	5,89	4,35 (73,91%)
	2	17,10	15,87 (92,81%)	7,08	4,35 (61,49)%

Table 2. Impact of Computation Time in $1\mathcal{D}$ and $2\mathcal{D}$ meshes

This configuration achieves to balance the degree of partitioning, the size of the problem and the communications involved. While for the same partitioning schemes running on 16 processors (`16n-1c`), the communication times begin to impact. Consider the case 1024x1024. For $\mathcal{D} = 1$, 16 processes must exchange 1024 cells with each of its 2 neighbors in the mesh (East/West), while for $\mathcal{D} = 2$, the 16 processes must exchange only 256 cells with each of its 4 neighbors (North/South/East/West). However, as can be derived from Table 2, the time reported in communications from the 2D partitioning is greater than the 1D. This is due to the significant increase in network traffic that introduce neighborhood-based communications. When this happens, the speedup is limited by the cost of communications and network latencies. This fact, give us some possibilities for the development of the proposals presented below.

Lattice Size	$\mathcal{D}$	Mesh Size	4n-1c	Mesh Size	16n-1c
256x256	1	4	1,90	16	3,40
	2	2x2	2,20	4x4	4,31
512x512	1	4	2,93	16	6,45
	2	2x2	3,02	4x4	6,83
1024x1024	1	4	3,68	16	10,88
	2	2x2	3,74	4x4	9,04

Table 3. Speedup

4 Multicore Environments

In this section, we describe some modifications to the CA implementation given in section 3 in order to restructure the prototype code and exposing an abstracted view of the multicore cluster to the CA applications developer.

4.1 MPI Process placement

In a multicore cluster based on distributed memory system, the parallel execution of P tasks, can be carried out by using several combinations between nodes and cores per

node. Considering a homogeneous hardware environment - with a maximum number of nodes M and the same number of cores per node C- the node-core combination is composed by: $P = n * c$ where $1 \leq n \leq M$ and $1 \leq c \leq C$. This fact involves taking a decision about what node-core combination delivers the best performance, through the evaluation of the key features of the algorithm that can affect - positive or negatively - the expected performance.

According to the previous issue, for the CA model implementation to exploit the underlying hardware, the MPI processes have to be placed carefully on the cores of the multicore cluster. Whilst MPI standard is architecture-independent, it is responsibility of the each implementation to bridge the gap between the hardwares performance and the applications.

Fortunately, recent MPI-2 implementations such as Open MPI [4] or MPICH2 [7] are able to take advantage of multicore environment and offer a very satisfactory performance level on multicore architectures. In particular, MPICH2 library is able to use shortcuts via shared memory in this case, choosing ways of communication that effectively use shared caches, hardware assists for global operations, and the like.

In a cluster of multicore nodes, it seems natural to employ a hybrid programming model where MPI and thread work together. MPI-2 provides a special init routine (MPI_Init_thread), to signal that you want to use MPI in a multithreaded environment [8].

However, there is always the option to use pure MPI and treat every CPU core as a separate entity with its own address space. The next experiments were carried out linking the skeleton to MPICH2. The Fig. 3 shows the performance of 1- and 2-dimensional partitioning of lattices of 256x256, 512x512 and 1024x1024 cells when the degree of partitioning applied to each lattice was 4 (`1n-4c`), 16 (`4n-4c`) and 64 (`16n-4c`).

In this experimental case all cores of assigned nodes are usable by the MPI processes with the restriction that only a single MPI process runs on a given core. The experiments consider two different MPI process launching policies. The first policy uses the four cores per node in base to a simple sequential ranking.

The last experiment, `32n-2c`, represents sixty-four processes on a 2D-mesh (8x8) using a round robin ranking. In this case, the mesh was made up of two cores from each of the 32 available nodes in the cluster.

In the case of sequential placement policy, we observed that as the degree of parallelism grows, the performance improves. However, the better performance is achieved when we do not fully use all the cores in a node (32 nodes under-subscribed). This configuration effectively provides more memory bandwidth to each core and improves the network latency experienced by each core, but it is not recommended because running with fewer than the maximum number of cores per node reduces overall throughput of a computing cluster.

Besides the problem to assign processes to nodes, it comes other problem related to the distribution of processes to specific cores inside a single node. We observed that the default policy used by MPI not all distributions are able to establish automatically an affinity mechanism between processes and cores.

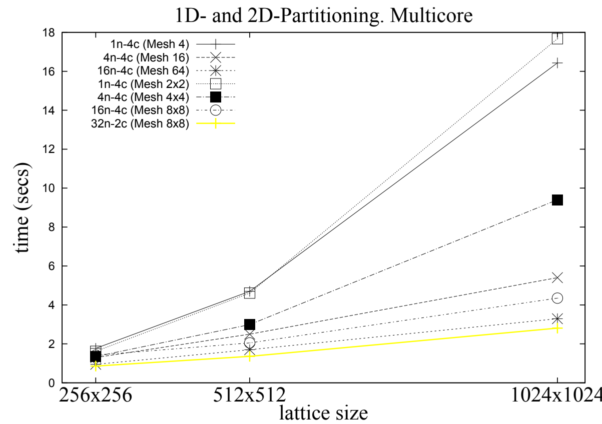

Fig. 3. $1\mathcal{D}$ and $2\mathcal{D}$ Partitioning. Multicore Execution Times

On Linux Operating System, the system call `sched_setaffinity` has the ability to specify in which core within the node a certain process will execute. We incorporate in the skeleton this facility based on the knowledge of the hierarchy of multicore cluster.

The Fig. 4 shows the execution time of the CA skeleton as a function of lattice size, applying an explicit affinity between those neighboring processes that exchange data and that have been allocated in the same node. As expected, there are vast reductions in the execution times, showing an average reduction of the order of 13%, 25% and 30% for `1n-4c-Aff`, `4n-4c-Aff` and `16n-4c-Aff` experiments respectively.

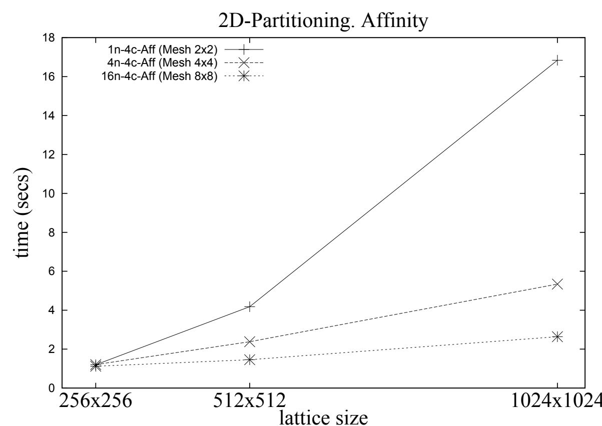

Fig. 4. $2\mathcal{D}$ Partitioning. Multicore Execution Times using Affinity

Note that for this case of analysis (D=2 and $\mathcal{D} = 2$), the CA skeleton was designed to allocate, in matrix notation, the sublattice (ij) to the MPI process with rank $j + i * P^{1/D}$. In sequential order, ranks $0...3$ go to the first node, ranks $4...7$ to second node, and so forth. In round robin order, the MPI process rank 0 goes to the first node, rank 1 to the second node, and so forth. In no case, the CA model topology maps efficiently to the hardware topology. This leads to design a new policy to distributing sublattices to MPI process, which is explained in the next section.

4.2 The Mapping Problem

By applying different strategies in the prototype, either in the skeleton code (e.g., affinity) as in its execution environment (MPICH2), the skeleton implementation has achieved a considerable reduction in its execution time.

No matter of use case, a relevant observation is that the parallel implementation of the CA model, includes stable communication patterns in which data interchange occurs among neighboring sublattices. However, the methodology of allocating work to the MPI processes does not regroup sublattices on the same node as much as possible in a way that it reflects the behavior based on neighborhood of the CA model. This can be observed graphically in Fig. 5 (left), for a 2D-Mesh (8x8). All nodes have the same setup, for example, sublattices 00, 01, 02 and 03 are assigned to node 0, sublattices 04, 05, 06 and 07 to node 1 (not showed in the figure), sublattices 10, 11, 12 and 13 to node 2 and so on. Each node must manage ten inter-node communications (marked with deep blue), two intra-node (dark blue) and four inter-chip (light blue).

The Fig. 5 (right) shows the configuration when the CA model implementation applies a new mapping of sublattices to the different processes. In order to minimize inter-node communication, a new mapping strategy is performed based on knowledge of the communication mesh. This new mapping accomplishes two objectives: (1) all nodes manage the same amount of inter-node communication and (2) it takes advantage of multicore nodes hierarchy. As can be seen in the figure, of the four neighbors of a sublattice, one of them is mapped on the same chip with which it shares the cache, the other is mapped on the same node with which it shares the memory and the exchanges with the other two neighbors inevitably require inter-node communication.

For CA applications, the mapping was defined as a function of the indices of the data partition (sublattices) and since there are no cross dependencies, the mapping becomes relatively easier. Through this mapping, performance improvements were observed for our 2D-CA benchmark.

5 Results and Final Remarks

In a previous work, we have presented an implementation of parallel CA skeleton. It attacks the classical problems inherent to parallel programming such as task and data decomposition and communications. The skeleton frees the non-expert user from the burden of dealing with low issues of parallelism.

The emergence of multicore processors with shared caches and non uniform memory access causes the hardware topology to became more complex. Therefore, a real

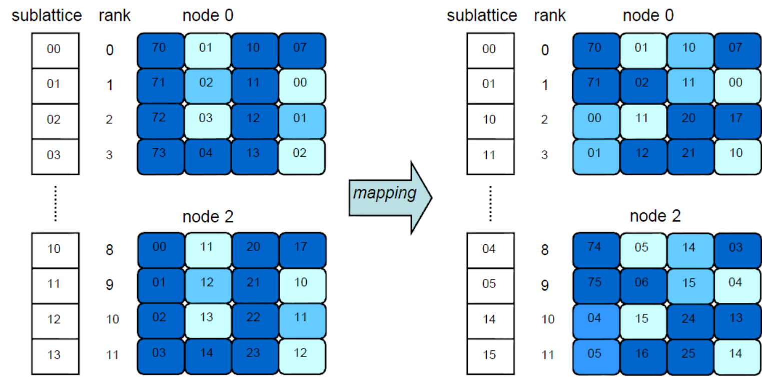

Fig. 5. Mapping of sublattices to MPI processes ranks in a 2D-Mesh (8x8)

challenge for parallel applications is to exploit such architecture at their potential. In order to achieve the best performance, many new factors must be considered and studied.

In this paper, we carried out experimental work that enabled us to understand the behavior of the architecture and implementation of the skeleton.This was possible because the parallel CA model is a typical application in which the data interchange occurs among neighbor sublattices and the communication pattern does not change across multiple executions.

The first experiment scattered the processes among the nodes. The second experiment regrouped the MPI processes on the nodes, but the operating system chose the placement of them among the cores. The third experiment also regrouped processes, but the skeleton implementation applied an affinity based on the knowledge of the hierarchy of multicore cluster. These experiments showed how the different MPI processes launching strategies impact in the performance on a multicore cluster and they were useful for exploring the hierarchy of the architecture.

Afterwards, we show a new mapping strategy that can obtain benefit in the performance by adapting its communication structure to the hardware affinities among processes.

The Fig. 6 shows a comparison of the implemented strategies in this work considering tree different lattice sizes. For the experiments called `-Aff` and `-Aff-Mpp`, the MPI processes were allocated on the same node as much as possible, i.e. in sequential order. For a small mesh size as 2x2 (partitioning degree=4) , the multicore strategies performed in much the same way as the non multicore one (`n-1c`). This behavior illustrates how the high costs of sequential computation in each core can not support optimizations. For 16 and 64 cores, the simultaneous application of all multicore strategies are the cases that achieve better performance.

The experimental work of this research was showing guidelines that are important to consider to run applications on a multicore system. With respect to its former version, this CA prototype release introduced features that significantly improved performance of the simulation of topologically connected problems.

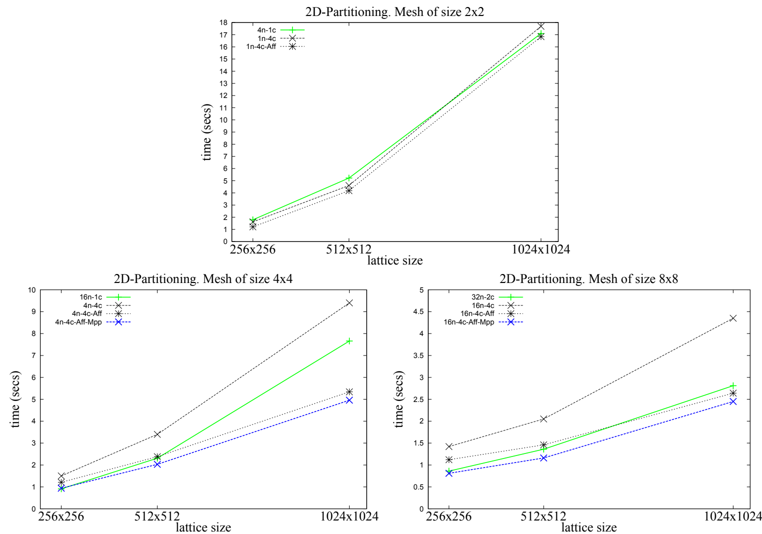

Fig. 6. Execution Times using 4 (top), 16 (bottom-left) and 64 cores (bottom-right)

6 Future Work

We have implemented a first version of the high performance 2D-CA skeleton and early experiences showed us that the strategy of allocation is very important in a multicore environment. But, all developments were performed on a particular type of cluster, of quadcore nodes. We are working to generalize the mapping strategy to cluster with larger number of core per node, where the hierarchy of memory access can be even greater.

The skeleton now includes an additional module to perform a mapping of processes at runtime. The design of this module requires three types of information: a) size of problem and partitioning degree chosen by the user, b) partitioning scheme applied to CA and c) topology of the parallel machine (cluster). The first two requirements are supplied directly by the CA skeleton `(N,P)` and (`D` and $\mathcal{D}$) respectively. Gathering the hardware information is essential to perform the mapping the mesh of communication in the resources of cluster assigned. While this experimental work applied a strategy of mapping in knowledge of the underlying architecture of the cluster, we are studying a tool that gives support to obtain the third piece of information required by our mapping module. This tool should provide the information of the hardware features such as number of cores per node, connectivity, cache size of the nodes allocated for parallel execution of the CA, among others. In addition information is required on the identification of processing elements (cores) available for allocation of sub lattices. With some differences, recent publications [6] [2] agree to propose a system API or runtime system to obtain the topology information in a portable way and export it to mapping module transparently to the user.

We are also examining other factors that probably influence the performance of communications, as cache and shared memory sizes.

Acknowledgments

We wish to thank the Department of Computer Architecture and Operating Systems of the Universidad Autonoma of Barcelona for allowing us to use their resources. Also to the Universidad Nacional of San Luis, the ANPCYT and the CONICET from which we receive continuous support.

References

1. OpenMP architecture processing reference model. ITU-TX.901,ISO/IEC 10746-1. available at http://enterprise.shl.com/RM-ODP/default.html.
2. François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2010)*, pages 180–186, Pisa, Italia, February 2010. IEEE Computer Society Press.
3. L. Chai, A. Hartono, and D. K. Panda. *Designing High Performance and Scalable MPI Intranode Communication Support for Clusters*. The IEEE International Conference on Cluster Computing (Cluster 2006), 2006.
4. Open MPI: Open Source High Performance Computing. http://www.openmpi.org.
5. Saez F. and Printista M. *Parallel Cellular Computing Model*. Proceedings of the IADIS international conference. ISBN: 978-972-8924-97-3 ,Vol 2, pages 145-149, 2009.
6. G. Mercier and J. Clet-Ortega. *Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments*. PVM/MPI 2009: 104-115, 2009.
7. MPICH2. http://www.mcs.anl.gov/mpi/.
8. R. Rabenseifner, G. Hager, and G. Jost. *Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes*. In Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009),427436, Weimar, Germany, 2009.