Towards a High Performance Cellular Automata Programming Skeleton

A. Marcela Printista and Fernando Saez

Universidad Nacional de San Luis.

Ejército de los Andes 950, San Luis, Argentina.

e-mail: {**mprinti@unsl.edu.ar, bfsaez@unsl.edu.ar**}

August 3, 2010

Abstract

Cellular automata provide an abstract model of parallel computation that can be effectively used for modeling and simulation of complex phenomena and systems. In this paper, we start from a skeleton designed to facilitate faster D-dimensional cellular automata application development. The key for the use of the skeleton 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 paper, we described some optimizations to restructuring the prototype code and exposing an abstracted view of the multicore cluster to the high performance CA application developer. The implementation of lattice division functions establishes a partnership relation among parallel processes. We propose that this relation can efficiently map on the multicore cluster communicational 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: **Skeletal Programming, Cellular Automata, Multicore Nodes, Mapping Strategy**

1 Introduction

Traditionally, parallel programs are designed using low-level message passing libraries, such as PVM or MPI. Message passing provides the two key aspects of parallel programming: (1) synchronization of processes and (2) communications between processes. However, users still encountered difficulties because these interfaces force to deal with low-level details, and their functions are too complicated to use for a nonexpert parallel programmer. Many attempts have been undertaken to hide parallelism behind some kind of abstraction in order to free the programmer from the burden of dealing with low issues.

A alternative is to provide a set of high-level abstractions which provides support for the mostly used parallel paradigms. A programming paradigm is a class of algorithms that solve different problems but have the same control structure. Parallel programming paradigms usually encapsulate information about useful data and communication patterns, and an interesting idea is to provide such abstractions in the form of programming templates or skeletons [6]. The essence of the programming methodology based on skeletons is that all programs have a parallel component that implements a pattern or paradigm (provided by the skeletons) and a specific component of an application (in charge of the user). A parallel skeleton encapsulates the control and communication primitives of the application into a single abstraction and frees the user to consider low-level details involved in high performance computing.

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.

Today, multicore processors are an option to become part for clusters of multiple sizes. These multicore processors contain multiple execution cores on the same chip, each of which can independently perform operations, thereby introducing a new level of parallelization to clusters. These kind of processors have emerged as a feasible alternative to use in high performance computing. As we explain later, many new factors must be considered and studied to achieve the best 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 relevent process placement.

There are a substantial amount of work in the fields of parallel CA, with work done in both generalized tools and tools dedicated to particular applications. Several high-level parallel programming systems like CAMEL [12], CAM [11], StarLogo [9], and CAPE [3] made possible development of parallel software abstracting from the parallel architecture on which programs run. Actually, CAMEL has its own programming language CARPET [10] that allows the programming of cellular automata algorithms.

The rest of the paper is organized as follows. Section 2 gives a background about cellular automata and cellular algorithms. Section 3 discusses the CA skeleton and presents some experimental results of the CA implementation on a Cluster. Section 4 describes the multicore environments and discusses the MPI Process placement problem. We then introduce a new mapping strategy. Section 5 presents the comparative analysis of strategies and the conclutions are given. Finally, the future work is outlined en Section 6.

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 $\delta : S^v \to S$ 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 more complex case, the cells can have more different values. (It is even thinkable, that each cell has 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). Other types of boundary conditions may be modeled by using preset values of the cell values for the boundary nodes or writing unique update rules for the cells at the boundary. The neighborhood structure and its boundary condition depends on the application under consideration. The most common neighborhoods are called Von Neumann and Moore.

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.

2.1 Overview of Cellular Algorithms

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 Figure 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.

```
AutoCel(Lattice,steps)
init(Lattice)
for t = 1 to steps
      for i = 1 to N
           for j = 1 to N
                nextState(Lattice,i,j)
```

Figure 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. Therefore, multicomputers are the appropriate computing platform for the execution of CA models when real problems must be solved. In this approach, the update of cells is synchronized and executed simultaneously by all processing nodes. 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. A common 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.

3 High Performance Simulation for CA Models

Libraries, like PVM or MPI (low-level abstract models) give 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 this limitation and to increase the level of abstraction 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 directly they correspond with frequent parallel patterns. The programmer expresses parallelism using a set of basic predefined forms with solution to the mapping and restructuring problems.

Following this approach, Saez et al. [4] 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 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. As a first task of implementation 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 processors. The structure of divisions produced by the proposed scheme and the partnership relation established among processors give place to communication patterns that are topologically similar to a $Mesh$. This partnership is the responsible of the assimilation the communicational topology of a CA.

If $\sqrt[D]{P}$ is a natural number and N is multiple of P, then a D-lattice can be divided in P sublattices given place to a D dimensional Mesh ($D - Mesh$). For a D-lattice, meshes of different dimensions can be made (for example, an tree-dimensional lattice can be divided into sublattices of two or one dimension), but these are not relevant to the facts discussed in this paper.

P determines the number of divisions produced on the lattice, D defines the dimension of the Mesh of processors and $\sqrt[D]{P}$ is the degree of each dimension. The skeleton implementation assigns each sublattice to a processor and let the nodes update them simultaneously. Independently of the topology, each processor will be responsible of a D-sublattice $(\underbrace{\frac{N}{\sqrt[D]{P}} \times ... \times \frac{N}{\sqrt[D]{P}}}_{D})$, 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. The rules are described by a function built by the user. At the end of each step, after update all cells in the local sublattice, all processors interchange the updated border cells and a communication among its neighboring processors in a D-Mesh topology takes place.

To improve performance, the implementation uses asynchronous (non-blocking) communication calls available in MPI. Once the data communication is issued, the process can perform the computations of those cells that do not depend on the data expected from the neighboring processors in the D-Mesh. Finally, the process wait for the end of the communication. The iterative process is repeated as necessary.

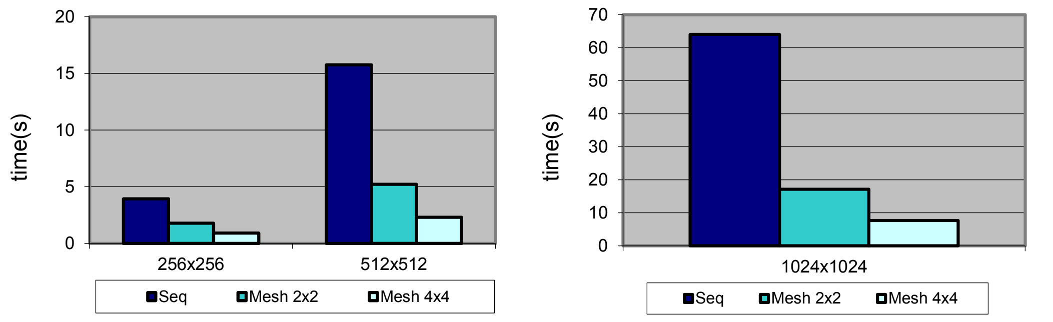
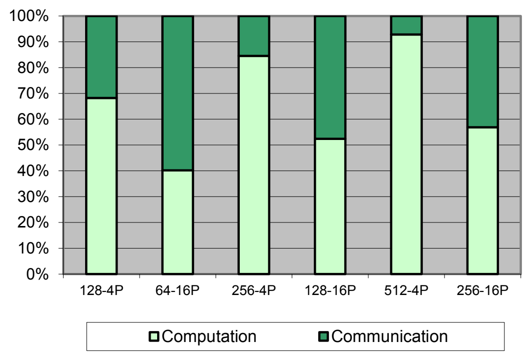


Figure 2: Execution Times on a cluster of 32 multicore nodes. One core by node



	Lattice size		
Speed Up	256x 256	512x512	1024x1024
4 Nodes	128x128	256x256	512x512
	2,20	3,02	3,74
16 Nodes	64x64	128x128	256x256
	4,31	6,83	8,36

Sublattice size

Figure 3: Computation-Communication relation. Speedup

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. The Figure 2 shows the experiments called `nx1c`. The references Mesh 2x2 and Mesh 4x4 correspond to 4 and 16 nodes, respectively. 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.

In addition to overall execution time, we measured the computation and communication times for each lattice size. The Figure 3 (left) shows that using 4 processors (Mesh 2x2), the computation ratios are much higher than the communication, 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 Figure 3 (right). This configuration achieves to balance the degree of partitioning, the size of the problem and the communications involved. While for the same partitioning scheme running on 16 processors (Mesh 4x4), the relation between computing and communications begins to match. 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.

4 Multicore Environments

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. From a point of view of programmers, pure MPI ignores the fact that cores inside a single node work on shared memory. Moreover, it is not required for the MPI library and underlying software layers to support multi-threaded applications, which simplifies implementation.

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. [5] 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 emply 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.

Affortunately, recent MPI-2 implementations such as Open MPI [2] 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 the following sections, 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 can 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 responsability of the each implementation to bridge the gap between the hardwares performance and the applications.

The next experiments were carried out linking the skeleton to MPICH2. The Figure 4 shows the performance of 2-CA for lattices of 256x256, 512x512 and 1024x1024 cells when the degree of partitioning applied to each lattice was 4, 16 and 64.

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. These experiments are called:

- `1nx4c`, to represent four processes on a 2-mesh of processors (2x2)
- `4nx4c`, to represent sixteen processes on a 2-mesh of processors (4x4)
- `16nx4c`, to represent sixty-four processes on a 2-mesh of processors (8x8)

The last experiment, `32nx2c`, represents sixty-four processes on a 2-mesh of processors (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

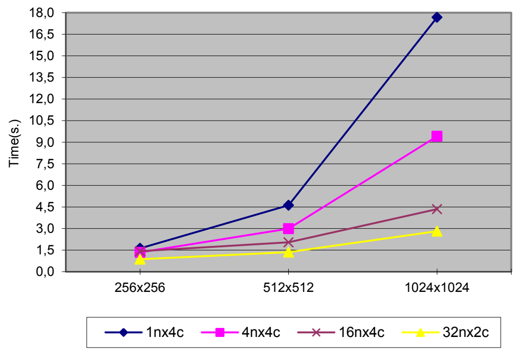


Figure 4: Multicore Execution Times

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.

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 incoporate in the skeleton this facility based on the knowledge of the hierarchy of multicore cluster.

The Figure 5 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 `4c-Aff`, `16c-Aff` and `64c-Aff` experiments respectively.

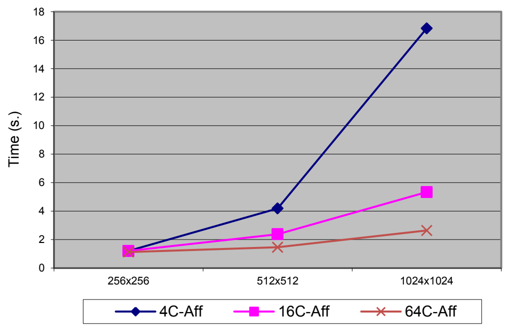


Figure 5: Multicore Execution Times
using Affinity

It is important to realize that the AC 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 $0 \ldots 3$ go to the first node, ranks $4 \ldots 7$ 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 any 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 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 Figure 6 (left), for a 2-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) and sublattices 10, 11, 12 and 13 to node 2. Each node must manage ten inter-node communications (marked with deep blue), two intra-node (dark blue) and four inter-chip (light blue).

The Figure 6 (right) shows the configuration when the CA model implementation applies a new mapping of sublattices to the different processes. 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 interchanges with the other two neighbors inevitably require inter-node communication.

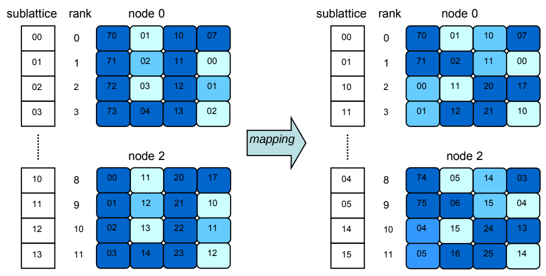


Figure 6: Mapping of sublattices to MPI processes ranks in a 2-Mesh (8x8)

5 Results and Final Remaks

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 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 pattern to the hardware affinities among processes.

Figures 7 and 8 show a comparison of the implemented strategies in this work considering tree different lattice sizes. For the experiments called `-Aff` and `c-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 (`nx1c`). 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.

6 Future Work

We have implemented a first version of the high performance 2-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. We are also examining other factors that probably influence the performance of communications, as cache and shared memory sizes

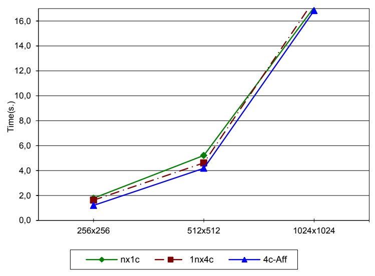
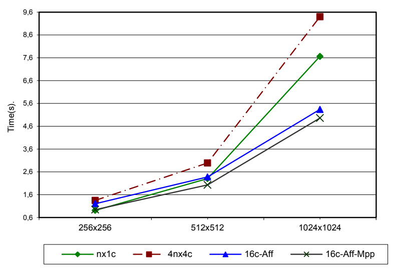


Figure 7: Execution Times using 4 cores



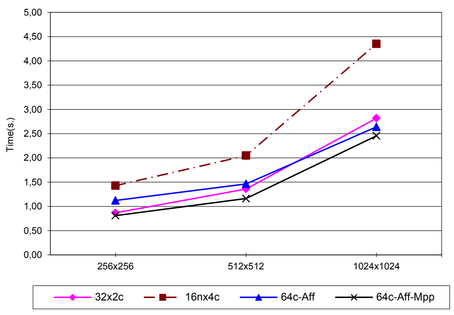


Figure 8: Execution Times using 16 (left) and 64 cores (right)

Acknowledgments

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

Bibliografa

[1] OpenMP architecture processing reference model. ITU-TX.901,ISO/IEC 10746-1. available at http://enterprise.shl.com/RM-ODP/default.html.

[2] Open MPI: Open Source High Performance Computing. http://www.openmpi.org.

[3] Norman M.G. Henderson J.R. Main G. Wallace D.J. *The use of the CAPE Enviroment in the simulation of Rock Fracturing*. Concurrency: Practice and Experience 3, pp.687, 1991.

[4] 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.

[5] A. Hartono L. Chai and D. K. Panda. *Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters*. The IEEE International Conference on Cluster Computing (Cluster 2006), 2006.

[6] Cole. M.I. *Algorithmic Skeletons: Structured Management of Parallel Computation. Research Monographs in Parallel and Distributed Computing.* Pitman, London, UK., 1989.

[7] MPICH2. http://www.mcs.anl.gov/mpi/.

[8] G. Jost R. Rabenseifner, G. Hager. *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.

[9] M. Resnick. *Turtles, Termites, and Traffics Jams*. The MIT Press, 1994.

[10] D. Spezzano G., Talia. *CARPET: A Programming Languaje for Parallel Cellular Processing*. In Proc. 2nd European School on Parallel Programming Environments (ESPPE 96), 1996.

[11] Margolus N. Toffoli T. *Cellular Automata Machines A New Enviroment for Modeling*. The MIT Press. Cambridge, Mass, 1987.

[12] Spezzano G. Talia D. Di Gregorio S. Rongo R. Spataro W. *A Parallel Cellular Tool for Interactive Modeling and Simulation*. IEEE Computational Science & Enginieering 3, p.33, 1996.