View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Servicio de Difusión de la Creación Intelectual

Parallel implementation of a Cellular Automata in a hybrid CPU/GPU environment

Emmanuel N. Millán[1,2,3], Paula Cecilia Martinez[2], Verónica Gil Costa[4], Maria Fabiana Piccoli[4], Marcela Printista[4], Carlos Bederian[5], Carlos García Garino[2], and Eduardo M. Bringa[1,3]

[1] CONICET, Mendoza
[2] ITIC, Universidad Nacional de Cuyo
[3] Instituto de Ciencias Básicas, Universidad Nacional de Cuyo, Mendoza
[4] Universidad Nacional San Luis, San Luis
[5] Instituto de Física Enrique Gaviola, CONICET
{emmanueln@gmail.com,ebringa@yahoo.com}
http://sites.google.com/site/simafweb/

Abstract. Cellular Automata (CA) simulations can be used to model multiple systems, in fields like biology, physics and mathematics. In this work, a possible framework to execute a popular CA in hybrid CPU and GPUs (Graphics Processing Units) environments is presented. The inherently parallel nature of CA and the parallelism offered by GPUs makes their combination attractive. Benchmarks are conducted in several hardware scenarios. The use of MPI /OMP is explored for CPUs, together with the use of MPI in GPU clusters. Speed-ups up to 20 x are found when comparing GPU implementations to the serial CPU version of the code.

Keywords: General purpose GPU, Cellular Automata, multi-GPU

1 Introduction

Multicore CPUs have become widely available in recent years. As an alternative, Graphics Processing Units (GPUs) also support many cores which run in parallel, and their peak performance outperforms CPUs in the same range. Additionally, the computational power of these technologies can be increased by combining them into an inter-connected cluster of processors, making it possible to apply parallelism using multi-cores on different levels. The use of GPUs in scientific research has grown considerably since NVIDIA released CUDA [1]. Currently, 43 supercomputers from the Top 500 List (June 2013, www.top500.org) use GPUs from NVIDIA or AMD. Currently the most commonly used technologies are CUDA [1] and OpenCL [2]. Recently Intel entered the accelerator's market, with the Xeon Phi [3] x86 accelerator, which is present in 12 supercomputers from the Top 500 List (June 2013, www.top500.org).

This work evaluates the trade-off in the collaboration between CPUs and GPUs, for cellular automata simulations of the Game of Life [4]. A cellular

automaton (CA) [5] is a simple model represented in a grid, where the communication between the grid points or cells is limited to a pre-determined local neighbourhood. Each cell can have a number of finite states which change over time, depending on the state of its neighbours and its state at a given time [6]. Even though such a model is simple, it can be used to generate complex behaviour. CA have been used to implement diverse systems in different fields of science: biological systems [7], kinetics of molecular systems [8], and clustering of galaxies [9]. CA have also been implemented in GPU architectures in biology to simulate a simple heart model [10], infectious disease propagation[11], and simulations of laser dynamics[12].

The Game of Life has already been extensively studied [13][14][15]. In this paper, the model is used as a starting point to developed future CA simulations in hybrid (CPU+GPU) environments, as it has already been achieved for Lattice Boltzmann Gas simulations [16], the Cahn-Hilliard equation [17][18] and reaction-diffusion systems [19] [20]. A relatively small code was developed to run in multiple parallel environments, and the source code is available in the website of the authors (`https://sites.google.com/site/simafweb/proyectos/ca_gpu`). Five implementations of the Game of Life code with C were developed: one serial implementation which executes in a single CPU core, and four parallel implementations, including: shared memory with OpenMP, MPI for a CPU cluster, single GPU, and Multi-GPU plus MPI for a CPU-GPU cluster. The paper is organized as follows. Section 2 describes the Game of Life in general; Section 3 contains details of the code implementation; Section 4 includes code performance; and conclusions are presented in Section 5, including possible future code improvements.

2 Description of the Problem

The Game of Life, through a set of simple rules, can simulate complex behaviour. To perform the simulations of this work, a 2 dimensional (2D) grid with periodic boundaries is used. The grid cells can be "alive" or "dead", and a Moore neighbourhood (8 neighbours) [6] is considered for each cell. According to the cell state at time t and the state of its 8 neighbours, the new state for the cell at time $t+1$ is computed. The update is done simultaneously for all cells , which means that their change will not affect the state of neighbour cells at $t+1$ time. Time t+1 is often referred to as the "next generation". The following rules define the state of a cell in the Game of Life[4]:

- Any living cell, with less than two living neighbours will die in $t+1$.
- Any living cell, with two or three living neighbours will live in $t+1$.
- Any living cell, with more than three living neighbours will die in $t+1$.
- Any dead cell, with exactly three living neighbours will be alive in $t+1$.

3 Implementation

3.1 Serial Code Implementation

The serial implementation which executes the entire simulation in one CPU processor was used to verify the correct functioning of other implementations. The serial code is quite simple. It begins initializing the main 2D grid, size NxN, with zeros and ones randomly placed. N has to be an even integer. A secondary NxN 2D grid is used to store the states of each cell for the next iteration. Any random number generator can be easily used, but the standard *rand(seed)* function was used in all codes. For testing purposes, the seed used to generate the random numbers is always the same. The code runs up to a maximum number of iterations defined by the user. Within each iteration, every cell and their neighbourhoods are monitored, the four rules for the CA are applied, and the next set of states are stored into the secondary array. Once all cells are analysed, the secondary grid is copied into the main 2D grid, concluding a given iteration. Every m iterations, with m a pre-set integer number, the state of the main 2D grid is written to an output file. Since this is a simulation with periodic boundaries, care must be taken when the values of the neighbours of a cell have to be monitored. When the simulation ends, the program prints on the screen the amount of RAM memory used and the total execution time, separating by: filling time of the 2D grid, evolution time, and write-up time of output files.

3.2 OpenMP Implementation

The parallel shared memory implementation of the code was developed with OpenMP [21]. It uses two compiler directives (pragma) to execute in parallel the *for* loops in the serial implementation of the code. These two loops are the loop that iterates over the 2D grid applying the game's rules, and the loop which copies the secondary grid to the main grid. Each *pragma* was configured to use dynamic or static scheduling, with the work assigned to each thread defined by the type of scheduling method. When using static scheduling, each thread is assigned a number of *for* loop iterations before the execution of the loop begins. When using dynamic scheduling, each thread is assigned some portion of the total (chunk) of iterations, and when a thread finishes its allotted assignment returns to the scheduler for more iterations to process.

3.3 MPI Implementation

Using a cluster of computers to run the case of message passing with MPI significantly increases the complexity of the code, especially when dealing with periodic boundary conditions. To handle part of the communications, the strategy by Newman [22] was used. The initial grid is loaded as in the serial case, then this grid is divided into equal blocks, with block size given by the full grid size and the number of processors in a square topology, e.g. running in 4 processors will give a 2x2 topology. Then, each block is sent to a different processor.

When an iteration begins, each process exchanges its grid boundaries with its neighbouring processes. From the code by Newman [22], the functions *exchange(), pack()* and *unpack()* were used as basis for sending the boundaries and corners of each block from each process to its neighbours. After performing the boundary exchange, the number of living neighbours for each cell in each process is calculated, the block corresponding to that process is updated, and a new iteration begins, sending updated boundaries between neighbouring processes. When it is necessary to copy the state of the grid to a file, the block from each process is brought to the master process to be part of a single grid, which is then written. Because of the simple chosen communication scheme, the current code cannot deal with odd number of processes.

3.4 GPU Implementation

The implementation of the code for a single GPU uses NVIDIA CUDA and takes the CPU serial code as basis. Two kernels (C code functions that run on the GPU) were developed: one kernel controls the state of the neighbours and modifies the secondary grid for the next iteration; the other kernel is responsible for updating the main grid with the data obtained by the first kernel (it copies the secondary grid into the main grid). There has to be a blocking step to ensure that all the threads have finished executing their instructions within the same kernel for a given iteration, making two separate kernels necessary. In order to ensure that instructions were properly executed, it was necessary to place a barrier outside the first kernel, in the CPU host code. Is worth mentioning that there is extra communication time: to copy the input grid generated in the CPU to the GPU, and to copy the system grid to the main memory each time the grid is written to a file, because it is necessary to copy from the GPU global memory to the CPU main memory.

3.5 Multi-GPU Implementation

The parallel code with MPI and CUDA is similar to the parallel MPI implementation explained in section 3.3 and the GPU implementation from section 3.4. MPI sends a block of the principal grid to each node for processing, then each of these nodes run the simulation on a GPU. Two kernels were added: the first kernel prepares data to be sent from a GPU to neighbouring CPU nodes, and the second kernel is responsible for loading data received from neighbouring processors into the GPU. These kernels replace the functions *pack()* and *unpack()* of the CPU MPI code. This multi-GPU code also outputs the time to copy data between the CPU and GPU at each time step, and it was called "Halo time" as in [23]. This only takes into account data transfer between CPU RAM memory and GPU global memory, regardless of MPI communication time between nodes

4 Benchmarks

4.1 Infrastructure

Simulations were executed in three different environments.

- Workstation Phenom: 2.8 GHz AMD Phenom II 1055t 6 cores with 12 GB DDR3 of RAM memory. NVIDIA Tesla c2050 GPU, with 448 CUDA cores working at 1.15 GHz, and 3 GB memory. Slackware Linux 13.37 64 bit operating system with kernel 2.6.38.7, OpenMPI 1.4.2, Cuda 5.0 and gcc 4.5.3.
- Workstation FX-4100: 3.4 GHz AMD FX-4100 x4 with 8 GB of DDR3 RAM memory. NVIDIA GeForce 630 with 48 CUDA cores working at 810 MHz,and 1 GB of memory. Slackware Linux 14 64 bit operating system with kernel 3.2.29, OpenMPI 1.7 beta, Cuda 4.2 and gcc 4.5.2.
- Cluster Mendieta at the Universidad Nacional de Córdoba: 8 nodes with two 2.7 GHz Intel Xeon E5-2680 CPU with 32 GB of memory, 12 NVIDIA Tesla M2090 GPUs with 6 GB GDDR5 (177 GBps) of memory and 512 1.3 GHz CUDA cores. The connection between nodes is at 20 Gbps InfiniBand DDR, with the switch using star topology. With Linux CentOS 6.4, MPICH 3.0.4 and Cuda 5.0.

Since the Phenom and FX-4100 workstations have only a single GPU, the code developed with MPI + GPU executes various MPI processes in the same workstation and executes the same number of independent processes in a single GPU. Because of this, the communication time between MPI nodes through the network is not evaluated for these two environments.

4.2 Simulation Results: Analysis and Discussion

Simulations were performed for different grid sizes, for the same number of iterations (1000) in all cases. The output of the last iteration performed for all parallel codes was checked against the serial version to verify the correct operation of the parallel codes. All codes were compiled with optimization -O3. Grids were NxN, with $N = 500, 1000, 2000, 4000$, and 6000. Four processors were used for the parallel implementations using OpenMP, MPI, and Multi-GPUs. Simulation times for different runs with the same configuration vary only by few percent. In figure 1 it can be seen the evolution of a small region of a particular simulation. In these frames it can be seen a complex pattern drawn from the simple set of rules that are part of the Game of Life.

The performance tests performed on the Phenom workstation can be seen in tables 1 and 2. For the smallest dimension (N=500), the code in OpenMP, MPI and GPU is always faster than the serial version. For the Multi-GPU code the cost of copying the "halo" between different GPU processes is high and performance degrades. For all other simulations, parallel codes are always faster than the serial code. The OpenMP code was executed in four independent threads and, therefore, it was expected that the simulation time would decrease approximately four times. This was not the case, because the parallelization with

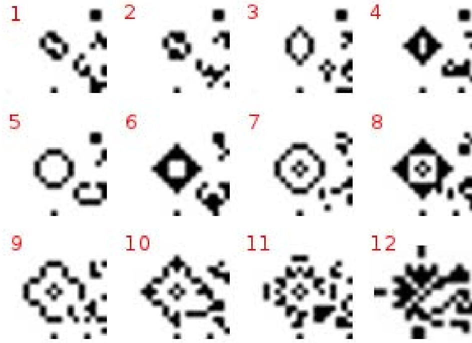

Fig. 1. Game of life simulation, for a grid with N=500, showing the evolution of a small region of the grid, for 12 different frames.

OpenMP was done by grid rows, and the use of the cache is far from optimal. The average speedups vs. the serial version, for all sizes, was 2.7 x. The implementation using MPI in 4 processors is approximately six times faster than the serial implementation, for all dimensions. The MPI code makes a division into blocks of the grid, and it is executed for the cases shown in tables 1 and 2, on a single workstation. Implementing OpenMP with blocks would likely increase performance. Speedups are given in table 5. When a single process is run on the GPU, the average speedup obtained for all grid sizes is 13 x. In the case of parallel Multi-GPU, the average speedup is 9.1 x. The copy time between GPU processes considering the smaller cases of the grid (500, 1000 and 2000) is greater than the computational time in each process. Therefore, running on a single GPU turns to be faster than using Multi-GPU for the grid sizes considered in this work. On the Phenom workstation (which has only one GPU), the Multi-GPU code executes multiple independent processes in the same GPU, at the same time. The largest case (N=6000) executes faster in multiple processes on the same GPU (Multi-GPU) than in a single process in the GPU, giving a 17 x speedup over the serial version. This improvement is due to the way in which the GPU scheduler administers the execution of the threads [24]. In addition, domain decomposition provides data locality, improving memory latencies [24] [25].

Tests performed on the cluster “Mendieta” were the same tests performed in the Phenom workstation. In the case of the cluster “Mendieta”, two nodes with two GPUs in each node were used. All tests were performed using one GPU per node, except for the largest case with N = 6000, which was also executed for two GPUs per node.

“Mendieta” was being shared with others users at the time of testing. This might be the reason why some execution times were greater than when using the Phenom workstation, which was used exclusively for these tests. Tables 3 and 4 contain the results. There are large differences in communication time between the multi-GPU case in Mendieta and the Phenom workstation for the transfer

Table 1. Simulation time in seconds for CPU serial, OpenMP and MPI codes, executing in the Phenom Workstation.

N	CPU Serial				OpenMP				MPI			
	fill	evolve	output	total	fill	evolve	output	total	fill	evolve	output	total
500	0.003	2.25	0.05	2.3	0.003	0.93	0.05	0.99	0.004	0.22	0.05	0.28
1000	0.01	10.03	0.17	10.21	0.01	3.44	0.19	3.64	0.02	1.15	0.21	1.37
2000	0.04	39.85	0.66	40.55	0.04	14.55	0.74	15.33	0.07	7.84	0.88	8.79
4000	0.16	167.15	2.64	169.95	0.16	56.75	2.95	59.86	0.25	30.23	3.47	33.95
6000	0.37	387.07	7.08	394.52	0.35	128.96	7.72	137.03	0.58	69.07	8.18	77.83

Table 2. Simulation time in seconds for GPU and Multi-GPU codes, executing in the Phenom Workstation.

N	GPU				Multi-GPU				
	fill	evolve	output	total	fill	evolve	output	halo	total
500	0.07	0.16	0.05	0.29	0.25	0.44	0.07	1.64	2.38
1000	0.09	0.56	0.2	0.86	0.26	0.77	0.24	1.65	2.93
2000	0.12	1.82	0.78	2.72	0.32	1.53	0.97	1.72	4.54
4000	0.26	7.35	3.21	10.81	0.49	4	4.06	2.99	11.53
6000	0.48	18.55	8.11	27.15	0.74	9.9	9.3	2.65	22.6

of information between GPUs ("halo" table column). When the GPU is used in the Phenom workstation, four parallel processes are executed in the same GPU, which causes an increase in the communication time in the PCI-Express bus. Using four GPUs for the larger case shows an improvement in the simulation time, decreasing it from 26.4 to 19.3 seconds. Speedups can be seen in table 6.

Table 3. Simulation time in seconds for CPU serial, OpenMP and MPI codes, executing in the Mendieta Cluster.

N	CPU Serial				OpenMP				MPI			
	fill	evolve	output	total	fill	evolve	output	total	fill	evolve	output	total
500	0.03	2.21	0.04	2.25	0.005	1.30	0.06	1.37	0.006	0.41	0.1	0.52
1000	0.01	8.94	0.16	9.11	0.01	5.25	1.07	6.33	0.03	1.22	0.41	1.66
2000	0.04	40.27	0.62	40.93	0.04	25.33	1.05	26.41	0.14	4.35	1.54	6.03
4000	0.16	188.49	3.78	192.42	0.27	101.05	4.23	105.54	0.59	26.66	7.01	34.26
6000	0.34	376.71	5.58	382.64	0.35	225.56	9.66	235.57	1.32	59.59	13.86	74.76

Tests were also performed on the FX-4100 workstation, for a single simulation, with *N=2000* and *1000* steps. Results are: Serial = 44.42 s, OpenMP = 18.63 s, MPI = 9.82 s, GPU = 17.5 s, Multi-GPU = 16.94 s. The video card installed in this machine is a low-range card and, as a result, its performance compare to CPUs is far from what could be reached with a high-range card.

Table 4. Simulation time in seconds for GPU and Multi-GPU codes, executing in the Mendieta Cluster.

N	GPU				Multi-GPU				
	fill	evolve	output	total	fill	evolve	output	halo	total
500	3.78	0.13	1.15	5.06	3.99	0.51	0.19	0.43	5.11
1000	3.81	0.43	0.31	4.55	4.08	0.53	0.42	0.55	5.59
2000	3.9	1.64	0.8	6.34	4.06	0.75	1.73	0.8	7.34
4000	4.09	5.66	2.98	12.73	4.3	1.32	6.77	1.28	13.67
6000	4.21	14.36	13.35	31.92	4.7	3.43	15.05	3.21	26.39
6000	running in 2 GPU per node				1.48	3.29	14.31	0.18	19.27

Table 5. Speedups for all parallel codes vs the serial code in the Phenom workstation

dim	Speedup vs Serial Code			
	MPI	OMP	GPU	Multi-GPU
500	8.32	2.33	8	0.97
1000	7.44	2.8	11.92	3.49
2000	4.62	2.64	14.91	8.94
4000	5.01	2.84	15.72	14.73
6000	5.07	2.88	14.53	17.46
AVG	6.09	2.70	13.02	9.12

Table 6. Speedups for all parallel codes vs the serial code in the Mendieta cluster

dim	Speedup vs Serial Code			
	MPI	OMP	GPU	Multi-GPU
500	4.33	1.64	0.45	0.44
1000	5.50	1.44	2.00	1.63
2000	6.79	1.55	6.46	5.58
4000	5.62	1.82	15.11	14.08
6000	5.12	1.62	11.99	14.50
6000 in 2 GPUs per node				19.86
AVG	5.47	1.62	7.20	9.35

5 Conclusions and future works

The implementation of the popular Game of Life [4] was used in this paper as a possible introduction to the problem of parallel processing of a CA on CPU+GPU hybrid environments. Improvements in the execution times in the pure GPU implementation and the Multi-GPU implementation have been achieved, with speed-ups approaching 20 x, when compared to a serial CPU implementation. The MPI implementation of the code gave better timing than the implementation using OpenMP, but the development time for the MPI code was higher. The OpenMP implementation did not perform as expected, and it would be necessary to carry out a detail code analysis with a tool like perf [26] to improve it. Using the Multi-GPU code provides significant speed-ups, but only for large grids (above a few million grid points).

There are several possible improvements for the codes presented here. For instance, the codes could be adapted to support square grids with N being an odd number, non-square grids, and odd number of processes. In addition, MPI and Multi-GPU codes deal with neighbours in a simple way, but MPI provides functions which could be used to perform these tasks more efficiently. There was no optimization of the codes, beyond the standard compiler optimization. There are several ways to optimize and improve performance in GPU codes: management of shared memory, monitoring and reducing the number of registers

used by each kernel, etc. Starting with CUDA 4.0, GPUDirect (http://goo.gl/g7D1p) technology is available and supported by MVAPICH [27], and data can be directly transferred between GPUs without passing through the main memory system.

Once an efficient CA is implemented in multiple GPUs, one could move to related problems, such as the Reaction-Diffusion Equations [19] [20], the Cahn-Hilliard equation [17][18], or a CA representing some general problem of interest [28].

6 Acknowledgements

E. Millán acknowledges support from a CONICET doctoral scholarship. E.M. Bringa thanks support from a SeCTyP2011-2013 project and PICT2009-0092.

References

1. CUDA from NVIDIA: http://www.nvidia.com/cuda
2. OpenCL, The open standard for parallel programming of heterogeneous systems: http://www.khronos.org/opencl/
3. Dokulil, J., Bajrovic, E., Benkner, S., Pllana, S., Sandrieser, M. and Bachmayer, B. Efficient Hybrid Execution of C++ Applications using Intel (R) Xeon Phi (TM) Coprocessor. arXiv preprint arXiv:1211.5530 (2012).
4. Gardner, M. The fantastic combinations of John Conway's new solitaire game "life". Scientific American 223 (October 1970) pp 120-123.
5. Wolfram, S. Cellular Automata as models of complexity. Nature Vol. 311. pp 419-424. 1984.
6. Ganguly, N., Sikdar, B.K., Deutsch, A., Canright, G. and Chaudhuri, P.P. A survey on Cellular Automata. Centre for High Performance Computing, Dresden University of Technology. 2003.
7. Alt, W., Deutsch, A., and Dunn, G., editors. Dynamics of Cell and Tissue Motion. Birkhuser, Basel, 1997.
8. Packard, N.H. Lattice Models for Solidication and Aggregation. In First International
9. Schonsch, B. Propagation of Fronts in Cellular Automata. Physica D, 80:433450, October 2002.
10. Caux, J., Siregar, P., Hill, D.: Accelerating 3D Cellular Automata Computation with GP-GPU in the Context of Integrative Biology. In: Salcido, A. (ed.) Cellular Automata - Innovative Modelling for Science and Engineering. InTech. ISBN: 978-953-307-172-5. 2011.
11. Arvizu, C., Hector M., Trueba-Espinosa, A., and Ruiz-Castilla, J. Autmata Celular Estocstico paralelizado por GPU aplicado a la simulacin de enfermedades infecciosas en grandes poblaciones. Acta Universitaria 22.6: pp. 16-22. 2012.
12. Lopez-Torres, M. R., Guisado, J. L., Jimnez-Morales, F., & Diaz-del-Rio, F. GPU-based cellular automata simulations of laser dynamics. Jornadas Sarteco 2012.
13. Adamatzky, A., ed. Game of life cellular automata. Springer, 2010.
14. Alaniz, M., Bustos, F., Gil-Costa, V., Printista, M. Motor de simulacin para modelos de Automata Celular. 2nd International Symposium on Innovation and Technology ISIT 2011. 28-30 Noviembre, Lima Peru. Noviembre 2011. ISBN: 978-612-45917-2-3. pp. 66-71. 2011.

15. Rumpf, T. Conways Game of Life accelerated with OpenCL. In Proceedings of the Eleventh International Conference on Membrane Computing (CMC11), p. 459. book-on-demand. de, 2010.
16. Tlke, J. Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA. Computing and Visualization in Science 13, no. 1: 29-39. 2010.
17. Cahn J.W. and Hilliard J.E. Free energy of a non-uniform system III. Nucleation in a two point compressible uid, J.Chem.Phys., vol. 31, pp. 688699, 1959.
18. Hawick, K. and Playne, D. Modelling and visualising the cahn-hilliard-cook equation. Tech. rep., Computer Science, Massey University, 2008. CSTN-049.
19. Smoller, J. Shock waves and reaction-diffusion equations. In Research supported by the US Air Force and National Science Foundation. New York and Heidelberg, Springer-Verlag (Grundlehren der Mathematischen Wissenschaften. Volume 258), 600 p. (Vol. 258). 1983.
20. Ready, A cross-platform implementation of various reaction-diffusion systems. https://code.google.com/p/reaction-diffusion/
21. The OpenMP API specification for parallel programming: http://openmp.org/
22. Newman, R.; MPI C version by Xianneng Shen. LAPLACE MPI Laplace's Equation in a Rectangle, Solved With MPI. http://people.sc.fsu.edu/~jburkardt/c_src/laplace_mpi/laplace_mpi.html
23. Lorna, S. and Bull, M. Development of mixed mode MPI/OpenMP applications. Scientific Programming 9, no. 2. pp 83-98. 2001.
24. Brown, W.M., Wang, P., Plimpton, S.J. and Tharrington, A.N. Implementing molecular dynamics on hybrid high performance computers short range forces. Computer Physics Communications 182. pp 898-911. 2011.
25. Millán, E.N., García Garino, C. and Bringa, E. Parallel execution of a parameter sweep for molecular dynamics simulations in a hybrid GPU/CPU environment. XVIII Congreso Argentino de Ciencias de la Computación 2012 (CACIC), Bahia Blanca, Buenos Aires. ISBN-978-987-1648-34-4. 2012.
26. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page
27. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/
28. Tissera, P., Printista, A. M., and Errecalde, M. L. Evacuation simulations using cellular automata. Journal of Computer Science & Technology, 7. 2007.