Computers and Electrical Engineering 48 (2015) 12–24

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Performance analysis of Cellular Automata HPC implementations

Emmanuel N. Millán [a,b,c,*], Carlos S. Bederian [d], María Fabiana Piccoli [e], Carlos García Garino [b], Eduardo M. Bringa [a,c]

[a] CONICET, Mendoza, Argentina
[b] ITIC, Universidad Nacional de Cuyo, Argentina
[c] Facultad de Ciencias Exactas y Naturales, Universidad Nacional de Cuyo, Mendoza, Argentina
[d] Instituto de Física Enrique Gaviola, CONICET, Argentina
[e] Universidad Nacional de San Luis, San Luis, Argentina

ARTICLE INFO

Article history:
Received 16 September 2014
Revised 17 September 2015
Accepted 18 September 2015

Keywords:
Hardware counters
Cellular Automata
High Performance Computing
Weak and strong scaling

ABSTRACT

Cellular Automata (CA) are of interest in several research areas and there are many available serial implementations of CA. However, there are relatively few studies analyzing in detail High Performance Computing (HPC) implementations of CA which allow research on large systems. Here, we present a parallel implementation of a CA with distributed memory based on MPI. As a first step to insure fast performance, we study several possible serial implementations of the CA. The simulations are performed in three infrastructures, comparing two different microarchitectures. The parallel code is tested with both Strong and Weak scaling, and we obtain parallel efficiencies of $\sim$ 75%–85%, for 64 cores, comparable to efficiencies for other mature parallel codes in similar architectures. We report communication time and multiple hardware counters, which reveal that performance losses are related to cache references with misses, branches and memory access.

1. Introduction

Cellular Automata (CA) are models composed of a lattice or grid of cells, where each cell has a given "state" which can change with time [1]. Most CA have a discrete time evolution, and the interaction among cells and the states of their neighbors define the next state a given cell is going to take. The two most used neighborhood models are the Von Neumann (4 neighbors for a 2D grid) and Moore (8 neighbors for a 2D grid) neighborhoods [2]. CA models have been used in several areas, from biology to image processing. They have been used to model pedestrian dynamics [3], ecological systems [4], and water flow [5]. They are also used as salient region detector [6], and for noise filtering [7]. One of the most popular CA is the John Conways Game of Life (GoL) [8], which displays complex behavior using simple interaction rules for its evolution.

Most CA implementations are serial, since that is enough to represent many systems of interest, but there is some work implementing CAs in parallel environments, including both distributed memory implementations and Graphics Processing Unit (GPU) implementations [9–11]. In the work by Rybacki et al. [12] several CA examples were tested (including GoL) on four different machines, with implementations for single core, multi-core and GPU. For GoL with a grid of 1000 $\times$ 1000 they obtained a

* Corresponding author at: FCEN, Universidad Nacional de Cuyo, Padre Jorge Contreras 1300, Parque General San Martín, CP M5502JMA, Mendoza, Argentina. Tel.: +54 261 4236003.
E-mail address: emmanuel.millan@gmail.com, emillan@itu.uncu.edu.ar (E.N. Millán).

http://dx.doi.org/10.1016/j.compeleceng.2015.09.015
0045-7906/

throughput of 0.77 steps/s for the serial implementation and 2.0 steps/s for a parallel version in four MPI processes (executed in a Core 2 Extreme Q9300 2.5 GHz with 8 GB of RAM). Szkoda et al. implements the Frisch–Hasslacher–Pomeau (FHP) CA algorithm for fluid flow modeling [13], with a CPU version using AVX, SSE, with threads support, and with CUDA (www.nvidia.com/cuda) for a GPU. They conclude that executing with SSE and 32 threads with four Xeon E5 4650L gives approximately 20% better performance that a single Tesla C2075 GPU. Tissera et al. developed a CA model to simulate pedestrian emergency evacuation, EVAC* [14], achieving ∼44% of speedup in eight MPI processes versus one MPI process, for a 2D grid of 200 × 125, with a communication time of approximately 45% of the total time, and concluding that simulated sizes were not large enough to justify the use of more MPI processes. CA are often used within the Lattice Boltzmann (LB) formalism, for instance to simulate fluid flow and transport [15]. In the work of Jelinek et al. [16] a large scale parallel LB was implemented in two dimensions using Fortran and MPI, performing reasonably in weak and strong scaling tests up to ∼40,000 cores. Pohl et al. [17] achieved ∼75% of parallel efficiency in 512 CPU cores, performing another LB simulation using a total of 370 GB of RAM. In the work of Coakley et al. [18] Agent-Based Model (ABM) [19] simulations were performed within the Flame framework achieving ∼80% of parallel efficiency in 432 CPU cores with the Circles benchmark for 500,000 agents. Rauch et al. [20] presented a parallel CA framework for the evolution of materials microstructure, and obtained a ∼10 × of speedup using 15 SGI Altix ICE 8200 nodes (for 27 million cells). Oxman et al. [21] developed three parallel implementations of the Game of Life CA: one shared memory implementation and two distributed memory implementations. They obtained the best results with the two distributed memory implementations.

Despite all the work devoted to High Performance Computing (HPC) implementations of CA, there is a need for a detailed study of performance using hardware counters, and this is the main objective of this work. In order to achieve this goal, we optimize a CA for HPC environments with multiple CPUs, using distributed memory (MPI) for fast simulation of large grids, which could be useful for problems like Reaction–Diffusion systems [22], or the particular case of Cahn–Hilliard equations [23], which are needed to model nanofoams [24]. We focus on the implementation of the GoL [8] CA, but more complex CA can be easily implemented using the optimized code developed here. We note that GoL is a data intensive, memory-bound problem, where parallelization by domain decomposition of the grid [25] is expected to be efficient due to the local nature of the interaction among cells. Most other CA are amenable to the same parallelization strategy, but their mathematical complexity might shift the relative importance of memory access, computation, and communication reported here.

We initially develop serial versions of the GoL CA, and then a parallel MPI version. A preliminary study on serial, OpenMP and MPI versions of GoL was already presented by Millán et al. [26]. Here, those serial and MPI versions have been significantly improved performing several optimizations, and we use hardware counters to gather information on code performance and detect bottlenecks. Hardware [27] and software counters allow fine-tuning of HPC applications, with a clear view of improvement or degradation in performance, and the ability to evaluate the behavior of a code through different events, such as stalls on the pipeline, branches miss-prediction, CPU migrations, context switching, instructions per cycle, memory access including cache references with misses, etc. [28–30].

This paper is organized as follows. Section 2 details the CA used in this work and gives a background of hardware counters. The Hardware Infrastructure is detailed in Section 2.1. The serial and parallel codes are described in Sections 2.2 and 2.3, respectively. Section 3 is divided in two subsections: in Section 3.1 the results obtained with the serial implementations are discussed, and in Section 3.2 strong and weak scaling are performed and discussed comparing the obtained results with the efficiency of two other mature and open source parallel codes: LAMMPS [31] for Molecular Dynamics (MD) [32], and Repast HPC [33] for agent-based-model (ABM) [19]. Finally, in Section 4 the conclusions and future work are covered.

2. Material and methods

The Game of Life (GoL) CA [8] uses the Moore neighborhood (8 neighbors) [2] and each cell of the grid can take two states, dead or alive. The evolution of the grid at each time step follows the following rules:

- Any living cell with less than two living neighbors will die in the next iteration (isolation).
- Any living cell with two or three living neighbors will live in the next iteration.
- Any living cell with more than three living neighbors will die in the next iteration (overcrowding).
- Any dead cell with exactly three living neighbors will be alive in the next iteration (reproduction).

We took the CA code developed in Millán et al. [26] as *Baseline* and implemented optimizations to the single CPU and multi-core with distributed memory versions (source code and data results are available from http://goo.gl/9X7tcy). GoL is simple to simulate: if a live cell is represented by a “1”, and a dead cell by a “0”, it only requires the sum of the states of the 8 neighbor cells to apply the CA rules described above. Therefore, it is not a computationally intensive CA, but results in a memory-bound CA. For this reason, we focus the analysis in memory related hardware counters. Because we are interested in general CA applications, for automata with more than two states and given by non-integer values, bit packing [34] is not implemented in our code.

The influence of the initial distribution of alive/dead cells in the evolution of Game of Life was studied by Gibson et al. [35]. They reported on neighborhood activity by counting the number of live cells that each cell has, with an initial distribution probability from 0% to 100%. In the range below 5% and above 80% they find that there is little to no activity after 1000 steps of simulation. Between 20% and 60% activity stays near its maximum value. Therefore, we perform all of our simulations with an initial distribution of ∼50% of live cells to reach this maximum activity level.

Hardware counters are present in CPUs as a set of registers that count events that occur in the CPU [27]. These events reflect how the code was designed, compiled and executed in the CPU. This relation can in turn be used to perform optimizations on the

source code. Browne et al. [27] discussed discrepancies in application performance and near peak performance in HPC machines, and to clarify them proposed to analyze processor performance metrics. Tinetti et al. [36] studied the use of hardware counters for different compiler optimizations, and they concluded that the causes of degradation in performance can be understood by using hardware counters.

Profiling of applications with hardware counters can be performed with several tools: Oprofile (oprofile.sf.net), vtune [37], perf_events [28], likwid [38] and PAPI [27]. In this work we use extensively *perf_events* (from now on perf), due to several reasons:

1. It is easily available because is included with the source code of the Linux kernel starting from version 2.6.31.
2. The user space tool *perf* provides a complete interface to hardware counters [28].
3. It is easy to use, because with a single command it can execute a number N of simulations, and outputs the average and standard deviation values for the selected counters
4. It provides hardware (cache misses, branches misses and CPU-cycles) and software events (such as context switching, page faults and CPU migrations).
5. *Perf* supports multiplexing of events, if the user specifies more events to count than available hardware counters, *perf* multiplexes the use of the hardware counters and gives each event a time slice to record their occurrences. At the end of the execution, *perf* shows the percentage of time each event was recorded, and scales the final count with Eq. (1) [39].

We notice that multiplexing of events can be misleading, when one event is being recorded, another important event could occur at the same time window and it will not be counted by *perf*. We do not use multiplexing of events here, but count each event 100% of the time.

$$final\_count = \frac{raw\_count * time\_enabled}{time\_running} \tag{1}$$

The hardware counters of interest in this case include cache references and misses, L1 data cache loads and misses, Last Level Cache (LLC) loads and misses, and branches with misses [30]. Optimizing the access to the Cache and lowering the amount of cache misses, generally provides a good starting point for optimizing code. Branches take an important role in code optimization: CPUs use a *pipeline* [29] to execute instructions, and when a *branch* is about to execute, the processor does not necessarily know which path the branch will follow (for example, in *if* conditional structures), until the instructions are executed. A module in the CPU known as *Branch Predictor* [29] is in charge of guessing which path branches will follow, and fill the pipeline with instructions from that path. If the wrong prediction was made (branch miss), the instructions executed are discarded and the pipeline has to be filled with the correct instructions from the path not taken previously. Therefore, lowering the number of branches present in a code, and the number of branch misses, will contribute to performance optimization.

In the next sections we introduce the main characteristics of hardware and software employed to execute the simulations, and hardware details for each microarchitectures are provided.

2.1. Hardware details and infrastructure

We executed the MPI and Serial simulations in three different infrastructures:

- Mini-cluster FX-8350 with 3 nodes (denoted as “FX-8350”), each equipped with: 4 GHz AMD FX-8350 (Piledriver microarchitecture) × 8 with 16 GB of DDR3 RAM memory. Slackware Linux 14.1 64 bit operating system with kernel 3.10.5, OpenMPI 1.8.1 and GCC 4.8.1.
- Cluster ICB-ITIC (denoted as “ICB-ITIC”) at the Universidad Nacional de Cuyo: two nodes with AMD Opteron 6272 (Bulldozer microarchitecture) CPU, with 64 CPU cores at 2.1 GHz, 128 GB of RAM and dual Gigabit Ethernet in each node. The Linux distribution installed in the cluster is Rocks Cluster 5.5, kernel 3.10.46, with OpenMPI 1.8.1 and GCC 4.8.1.
- Cluster Mendieta at the Universidad Nacional de Cordoba (denoted as “Mendieta”): 8 nodes with two 2.7 GHz Intel Xeon E5-2680 (Sandy Bridge-E microarchitecture) CPU with 32 GB of memory per node. The connection between nodes is at 20 Gbps InfiniBand DDR, with the switch using star topology. With Linux CentOS 6.4, kernel 2.6.32-358, OpenMPI 1.8.1 and GCC 4.8.2.

There are different processors in the different clusters used in our study: Xeon E5-2680, AMD Opteron 6272, and AMD FX-8350. A brief description of their architecture is given below. The Intel Xeon E5-2680 has 8 cores, each of them with two *threads* (Hyper Threading Technology), with a total of 16 *threads* per socket, and supports a maximum of two sockets interconnect by the QuickPath Technology [40]. However, Hyperthreading is disabled in “Mendieta”. It has three cache levels: the first level (L1) has 32KB instruction cache and 32KB data cache in each core, the second level (L2) has 256 KB shared instructions and data cache in each core, and finally an instruction and data Last Level Cache (LLC or L3) of 20 MB shared between all cores. It supports four DDR3 memory channels per socket. For a complete description of features of this processor see [40].

The AMD processors used in this work are based on two AMD microarchitectures: the AMD 6272 Opteron CPU is based on Bulldozer (also called AMD Family 15h) microarchitecture, and the AMD FX-8350 in an incremental update of Bulldozer, the Piledriver microarchitecture [41]. These two microarchitectures consist of “compute units”. The Operating System (OS) sees each *compute unit* as two different CPU cores. Each *compute unit* shares one floating point unit (FP), one first level instruction cache (L1 icache or IC) with 64 KB of memory, and one second level data cache (L2) with 2 MB. Within the *compute unit* there are two

integer cores with a dedicated x86 execution unit and a first level data cache (L1 dcache) with 16 KB. Also, a third level data cache (LLC or L3) with 8 MB is shared between all *compute units* present in the socket.

In the case of the FX-8350 there are four *compute units* (8 cores), and in the Opteron 6272 there are two modules of four *Bulldozer compute units* each one (with two cores in each compute unit), with a total of 16 cores (as seen by the OS) in a single socket. For a complete description of the architecture of AMD processors Family 15h see [41].

Regarding hardware counters, the AMD Family 15h has two types of counters: the first set is included inside the CPU core (six performance event counters, Sections 2.7.1 and 3.15 from AMD BKDG [41]), and the second set is in the *NorthBridge* (NB, four performance counters, Sections 2.7.2 and 3.16 from AMD BKDG [41]). This *NB* is part of the processor, not to be confused with the old configuration of the *Chipset, Northbridge* and *Southbridge*, which was a component of the motherboard and it was located outside the processor.

The list of supported hardware events is available from AMD [41] and from Intel [42]. It is important to note that the supported events vary for different processors and manufacturers. The events are identified by an *event code* number (in hexadecimal) and a *mask* (hexadecimal), for instance, the event L2_CACHE_MISS (for AMD Family 15h) has a code of *0x7e* and five possible *mask* values, *0x01* to identified Instructions fills, *0x02* for Data fills, etc. All this information is available from the AMD documentation [41]. The events included in the cores of the CPU can be passed to the *perf* command line tool with the following syntax: *# perf stat -e cpu/event=0x7e,umask=0x01/ command_to_execute*. To execute events present in the *Northbridge*, the *"cpu"* keyword has to be replaced with *"amd_nb"*. For instance, to measure the L3_CACHE_MISSES (code event *0x4e1*) for any core (umask *0xf0*) the following syntax can be used: *# perf stat -e amd_nb/event=0x4e1,umask=0xf0/ command_to_execute*.

The output of the command *showevtinfo* from the *libpfm* application (perfmon2.sf.net) can be used to see the entire list of hardware events supported by the system. The following is a list of the hardware events we monitored in the AMD Opteron 6272 CPU with partial information extracted from the *showevtinfo* command:

- DATA_CACHE_REFILLS_FROM_L2_OR_NORTHBRIDGE with event code *0x42* and the followings umask: *0x01* (Fill with good data) and *0x0f* (All sub-events selected).
- DATA_CACHE_REFILLS_FROM_NORTHBRIDGE with event code *0x43*.
- REQUESTS_TO_L2 with event code *0x7d* and the umask *0x5f* (All sub-events selected).
- L2_CACHE_MISS with event code *0x7e* and the umask *0x17* (All sub-events selected).

The events monitored for the *NB* were the following:

- CPU_IO_REQUESTS_TO_MEMORY_IO with event code *0xe9* and the followings umask: *0x98* (Local CPU to Remote Memory) and *0xa8* (Local CPU to Local Memory).
- READ_REQUEST_TO_L3_CACHE with event code *0x4e0* and umask *0xf7* (count any read request and measure on any core).
- L3_CACHE_MISSES with event code *0x4e1* and umask *0xf7* (count any read request and measure on any core).

These hardware counters will be used in Section 3.2.

2.2. Serial code implementations

We develop four serial implementations: *Baseline, Swap, One_grid* and *One_grid_ch*. The code from Millán et al. [26] is used here as the *Baseline* implementation. It can be considered a *naive* implementation which was not optimized, except from the optimizations provided by the compiler (*-O3*). The CA is implemented with periodic boundaries and neighbors are controlled with *if-like* statements (four lines of code) that are executed in each iteration inside each cell, and return the proper *(i,j)* coordinates. The second implementation is called *Swap*. The only change in this implementation is that it switches the arrays containing the next state with the current state. The periodic boundaries are controlled the same way as in the *Baseline* implementation.

The third implementation, called *One_grid*, uses a single grid or lattice to carry out the evolution of the CA: the current and next states are stored on the same grid. This implementation also adds ghost rows and columns [43], also called *halo*. By using the *halo*, the four *if-like* statements to control the periodic boundaries present in the *Baseline* implementation are no longer needed in the *One_grid* implementation. Neighbor searches can be optimized with several strategies. For instance, bit operations where used to find the nearest neighbors for a CA running in a hypercube [44]. In the *One_grid* code, the grid is incremented by *3* in each direction $(N+3) \times (M+3)$, as shown in Fig. 1 for two iterations with this optimization. The result of the evolution of a cell is stored in the diagonal *left-up* cell (cell $1'$ in the first iteration, second panel in Fig. 1). The cells of the grid are read from top to bottom and left to right. When reading the states of the neighborhood of a cell, the diagonal *up-left* cell is only needed by the current cell. When all the cells in the first iteration are computed, the next iteration needs to start from the shifted grid stored in the previous iteration, but changing the starting point and the direction. The second iteration starts from the *bottom right* corner (cell $9'$ in Fig. 1, iteration 2) and the cells are read from bottom to top and from right to left. The results from the second iteration are stored in the diagonal *right-down* cell.

The last serial implementation, *One_grid_ch*, is the same as *One_grid*, but uses the *char* data type instead of *int*. This gives 4 times less memory use. We are interested in complex CA with more than 2 states per cell, and for this reason we do not use bit packing [21,34].

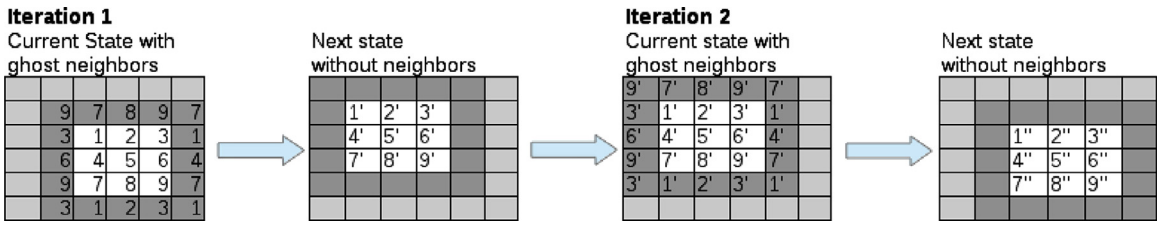

Fig. 1. *One_grid* code optimization. The original grid size is 3 × 3 (white background), three rows and columns are added to account for ghost neighbor (dark gray background) and the displacement needed to calculate the result of the next state (light gray background).

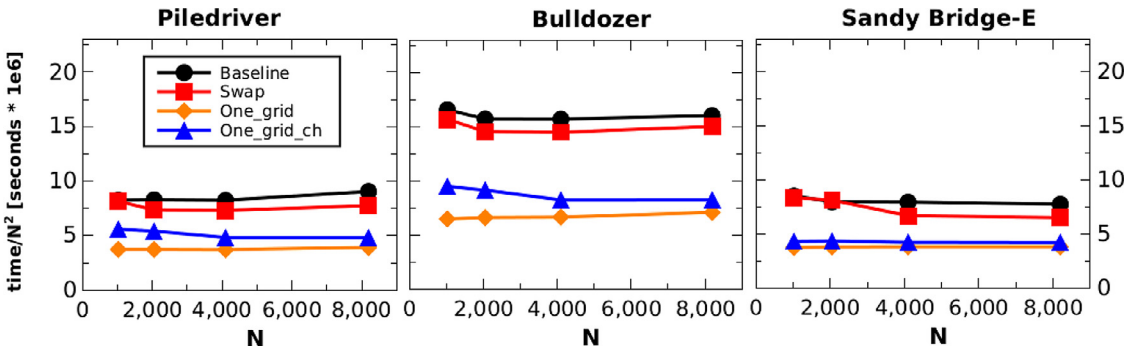

Fig. 2. Performance of CPU serial code implementations for different architectures, measured as wall clock time, execution of 1000 iterations for a grid size $N \times N$.

2.3. MPI code implementations

Three parallel implementations were developed: *Baseline, One_grid* and *One_grid_ch*. The *Baseline* MPI code used here includes some optimizations respect to the one presented in Millán et al. [26]. The code uses domain decomposition: each MPI process receives a block of the grid to evolve, then communicates asynchronously the border rows and columns to neighbor processes and starts evolving the center cells. When the MPI process finishes this, it controls that the border rows and columns were received from neighbor processes and then evolves the borders. Communications with neighbors are overlapped with processing of the cells from the center of the grid. The second parallel implementation, denoted as *One_grid*, uses the same concept as the serial *One_grid* implementation, with a single grid or lattice used to store the CA. Another difference with the *Baseline* MPI code is that this code cannot overlap communication with compute time. The third MPI implementation is similar to the second one, but using the *char* data type instead of the *int* data type (similarly as *One_grid_ch* from the serial code).

In previous work by Millán et al. [26] we tested a naive shared memory implementation of the Game of Life with OpenMP. We did not obtain positive results when comparing the performance of the shared memory implementation with a distributed memory implementation (MPI). Oxman et al. [21] conclude that the shared memory implementation of the Game of Life CA is easy to develop but does not scale well as more processors are added. They obtained their best results with a distributed memory implementation. This agrees with the preliminary results from [26], supporting the development of our parallel implementation with a distributed memory scheme.

3. Results and discussion

In this section we discuss the results obtained with the serial (Section 3.1) and parallel (Section 3.2) implementations. Each result from the CA simulations presented here is an average of ten simulations. Standard deviation is small, generally less than 1%, and for this reason, the error bars are not included in the figures. Additional details are given in the *Supplementary Material*.

3.1. Serial CPU implementations

The four serial implementations (*Baseline, Swap, One_grid* and *One_grid_ch*) are executed for four different square grid sizes, $N \times N$, with $N = 1024$, 2048, 4096 and 8192. All implementations are compiled with *-O3* compiler optimizations. In this section we name the clusters by their microarchitecture: *FX-8350* is denoted as *Piledriver, ICB-ITIC* as *Bulldozer* and *Mendieta* as *Sandy Bridge-E*. Fig. 2 shows the performance of the CPU serial code in these three infrastructures. Wall clock times (in seconds) are normalized with the total number of cells for each value of *N*, such that good performance would give a roughly constant normalized time as *N* increases.

The Bulldozer CPU has the lowest performance, which is expected since this CPU runs at the lowest clock rate, 2.1 GHz. The performances of the Piledriver and the Sandy Bridge-E microarchitectures are similar, even when the AMD (Piledriver) processor

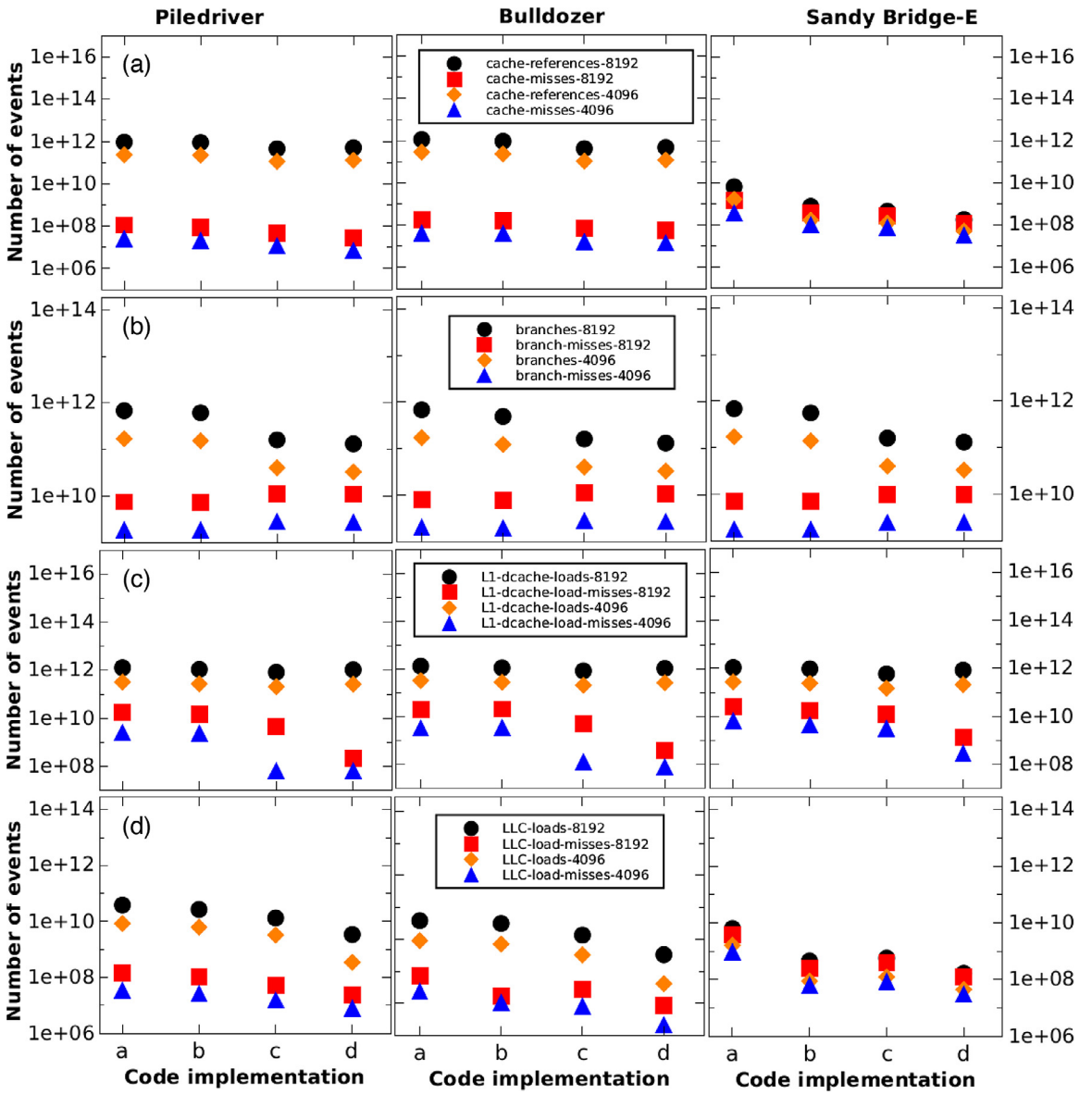

Fig. 3. (A) Cache references with misses, (B) Branches with misses, (C) L1 data cache loads with misses, and (D) Last Level Cache (LLC) with misses. Analyzed with perf for two different grid sizes, $N = 8192$ and $N = 4096$ in the three microarchitectures. Implementations: a = *Baseline*, b = *Swap*, c = *One_grid* and d = *One_grid_ch*.

runs at 4 GHz and the Intel (Sandy Bridge-E) processor runs at 2.7 GHz. The reason for this result could be the number of cache references (Fig. 3A) is much lower for the Sandy Bridge-E CPU than for the Piledriver processor. The same behavior can be seen for the Last Level Cache references (LLC) in Fig. 3D. The similarities in microarchitecture (Piledriver and Bulldozer) between the two AMD processors can be seen in the results from the Cache references, Branches, L1 data cache loads and LLC cache loads figures (from Fig. 3A–D).

The speedup between the *Baseline* and the *Swap* implementations is minor, approximately between a 1.12 × and 1.16 ×. The best serial CPU implementation is *One_grid*, which gives a ∼2.2 × speedup versus the *Baseline* code for $N = 8192$ in Piledriver. The last implementation (*One_grid_ch*) uses the *char* data type, using 4 times less memory than the *One_grid* implementation which uses the *int* data type. However, this decrease in memory use does no benefit the execution time of the CA compared with the *One_grid* implementation. This is due to *Partial-Register Writes* (see Section 2.19 and 4.8 of [45]). When memory footprint is not an issue, AMD recommends not to use data types of 16-bit or 8-bit in 32-bit or 64-bit software. The *One_grid_ch* implementation, in the best case, loses around a 12% in performance versus *One_grid*, and in the worst case loses about 40% in performance.

After presenting the detailed behavior of the serial implementations, we can now show the behavior of the parallel implementations.

Table 1
Strong scaling in the ICB-ITIC cluster for $N = 8192$ and 1000 steps. Time in seconds and speedup for N MPI processes versus 1 MPI process.

MPI processes	*Baseline*	*One_grid*	Speedup *Baseline*	Speedup *One_grid*
1	549	443	1	1
2	293	234	1.8	1.8
4	159	129	3.4	3.4
8	93	77	5.8	5.7
16	63	51	8.6	8.6
32	45	38	11.9	11.5
64	51	41	10.6	10.5

Table 2
Strong scaling in the Mendieta cluster for $N = 8192$ and 1000 steps. Time in seconds and speedup for N MPI processes versus 1 MPI process.

MPI processes	*Baseline*	*One_grid*	Speedup *Baseline*	Speedup *One_grid*
1	464	411	1	1
2	241	212	1.9	1.9
4	127	113	3.6	3.6
8	77	63	6	6.4
16	51	39	9	10.3
32	40	29	11.3	13.9
64	44	25	10.4	16.1

3.2. Parallel MPI implementations

Since we are interested in general CA, and the serial *One_grid_ch* (with the *char* data type) did not have good performance, the parallel *One_grid_char* implementation will not be included in the parallel results. *Strong* and *Weak* scaling tests were performed only for the parallel *Baseline* and *One_grid* implementations. In strong scaling, the size of the grid is fixed and the number of MPI processes is increased. In weak scaling, each MPI process receives the same amount of data, increasing the total size of the grid each time a new processor is added to the simulation. In order to be consistent, we always use the same code and only change the number of MPI processes (NP) as needed. This means that, when we report results for NP = 1, we are still using a parallel code, not a serial code from above. The execution times for the *One_grid* parallel code with NP = 1 are within ∼8% of the *One_grid* serial code execution times.

3.2.1. Strong scaling

We perform a strong scaling with $N = 8192$ during 1000 steps in the ICB-ITIC and Mendieta clusters with two parallel implementations, *Baseline* and *One_grid* (Tables 1 and 2). To include MPI initialization time we compare the MPI code with NP = (2, 4, 8, 16, 32 and 64) versus the same MPI code with NP = 1. In the ICB-ITIC cluster, the best result is obtained running the one_grid implementation with 32 MPI processes with ∼11 × of speedup. For the Mendieta cluster, the best performance is obtained with 64 cores for the *One_grid* implementation, with a speedup of ∼16 ×. These speed-ups are clearly smaller from what would be expected for an efficient code. The main reason for the poor performance is that this is a relatively small grid for strong scaling of this problem, but a much larger grid would have increased significantly the cost of the many tests carried out with detailed hardware counters. Future testing would need to include such large grids.

Table 3 shows the speedups of the *One_grid* implementation versus the *Baseline* implementation for the strong scaling shown in Tables 1 and 2, with $N = 8192$ and 1000 steps. The *One_grid* implementation in the Mendieta cluster obtains greater speedups than in the ICB-ITIC cluster with NP > 16, a behavior also present in the Weak scaling performed and discussed in the next subsection. To compare the obtained speedups of our code in the strong scaling regime to a mature parallel code, we selected the Molecular Dynamics (MD) code LAMMPS [31] (also used in the next subsection). We executed the Lennard–Jones potential (LJ) benchmark [46] with 864,000 atoms and 5000 steps in the ICB-ITIC cluster, for NP = 64, and achieve ∼45 × of speedup with a parallel efficiency of ∼70 %. Our GoL implementation for a grid of 64,000 × 64,000 and 1000 steps achieves a ∼56 × of speedup with ∼87 % efficiency.

In the strong scaling performed in the work of Jelinek et al. [16] for a Lattice Boltzmann (LB) simulation they achieve ∼30% of efficiency running in 12 CPU cores in one node and the same efficiency in 48 CPU cores in four nodes. If they use only 2 CPU cores per node, they obtain a perfect efficiency (100%) up to 3072 CPU cores. The simulations were executed in the Kraken cluster, where each node has two six-core AMD Opteron "Istanbul" 2.6 GHz processor, located at Oak Ridge National Laboratory. In our tests in the ICB-ITIC and Mendieta clusters we obtain, for 32 processes, ∼35% and ∼43% parallel efficiency, respectively. It

Table 3
Speedup of *One_grid* versus *Baseline* implementation from execution times shown in Tables 1 and 2, with $N = 8192$ and 1000 steps.

MPI processes	ICB-ITIC speedup	Mendieta speedup
1	1.24	1.13
2	1.25	1.14
4	1.23	1.12
8	1.21	1.22
16	1.24	1.31
32	1.18	1.38
64	1.24	1.76

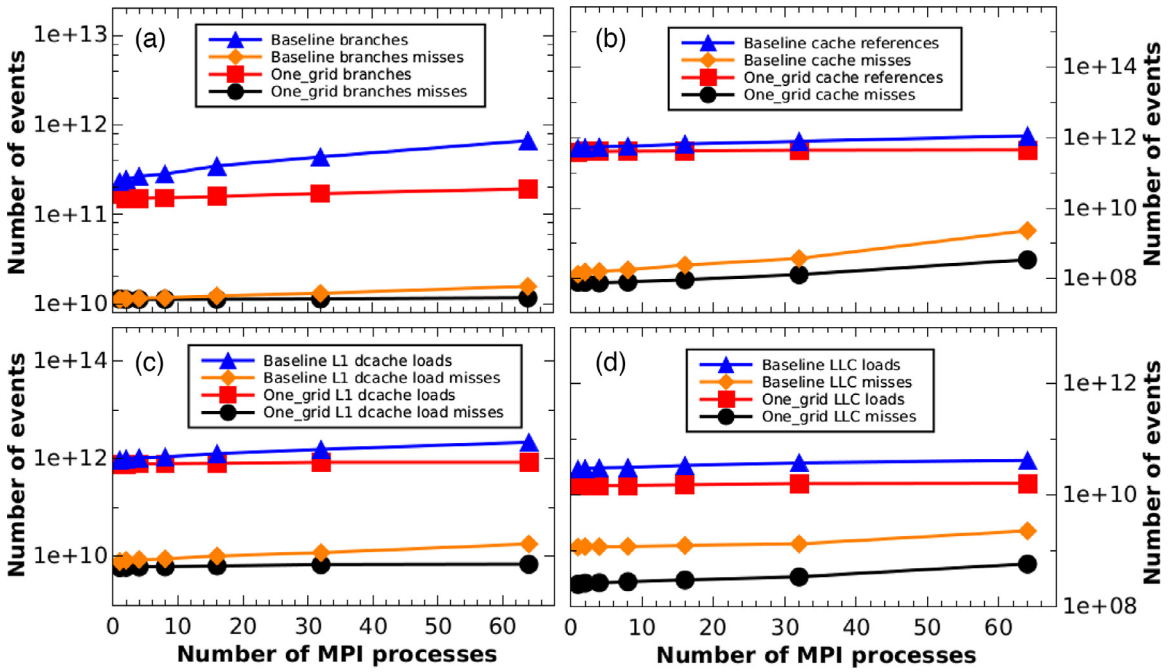

Fig. 4. Strong scaling (increasing number of processes while keeping the same system size) in the ICB-ITIC cluster for $N = 8192$. Number of events: (A) Branches with misses, (B) Cache references with misses, (C) L1 data cache loads with misses, and (D) Last Level Cache (LLC) with misses, versus Number of MPI processes (NP). Analyzed with perf for two implementations: *Baseline* and *One_grid*.

is important to note that the LB model is more complex than the Game of Life and requires a high memory bandwidth to perform well, which is why the LB simulation obtained a perfect efficiency only when using two CPU cores per node [16] and a lower efficiency when using all the available CPU cores per node.

Four groups of hardware events were tested in the ICB-ITIC cluster: branches with misses, cache references with misses, L1 data cache loads with misses and LLC loads with misses (Fig. 4A–D). Perf opens a file descriptor for each event and each child process of the *mpirun* command. Then, it counts the events of each MPI process individually, and finally reports the total sum of the events counted [39].

Fig. 4A–D all show a smooth increase in cache access and misses for all cache levels, with a much smaller increase for the *One_grid* MPI implementation. The fraction of misses is typically also much smaller for that implementation, supporting the better code performance.

We also perform an analysis of communication time using the profiler mpiP [47]. We tested the CA using half of the available RAM memory in one node of the ICB-ITIC cluster (~64 GB) with $N = 128{,}000$ during 1000 steps, with a processor grid of 8×8 (64 cores). The communication time of the evolution of the CA is ~21 % of the total simulation time (without output). For a similar simulation, only reducing the memory size by four ($N = 64{,}000$, ~16 GB), the communication time for evolution of the CA is ~8.5% of the total simulation time. Changing the processor topology to 4×8 (32 cores) and performing the same two tests, the communication times were ~4% and ~3 % respectively, indicating that the domain decomposition scheme is working reasonably well.

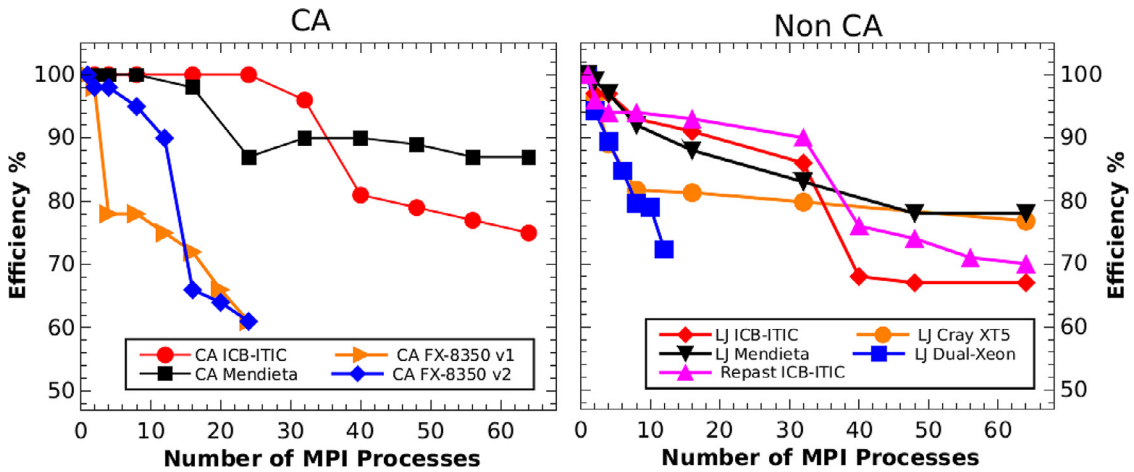

Fig. 5. Efficiency in weak scaling (increasing system size proportionally to number of processes) for three HPC codes: CA (this work), LJ LAMMPS [46] and Repast HPC [33]. See text for simulation details.

3.2.2. Weak scaling

We performed weak scaling calculations and compared the obtained results with two mature parallel codes, LAMMPS [31] and Repast HPC [33]. LAMMPS [31] is a Molecular Dynamics (MD) [32] code optimized to run in HPC clusters with up to hundreds of thousands CPU cores. We executed LAMMPS [31] using its Lennard–Jones potential (LJ) benchmark [46], with 32,000 atoms/processor. The LJ interaction among atoms is short-ranged, allowing a domain decomposition which should have an efficient parallel scaling. Regarding Repast HPC [33], we tested the example simulation "Zombie" included with the source code, which also includes "short-range" interactions which should scale well in a parallel environment.

Fig. 5 shows in the left panel the Game of Life CA *One_grid* implementation with a grid of 3000×2000 for each MPI process during 500 steps. This grid size (∼22 MB) was selected because it occupies more than the size of the L3 + L2 cache of the AMD Opteron 6272 and the Intel Xeon E5-2680. Bigger, square sized grids, like 4096×4096 (∼64 MB) would have taken more time to compute and the cost for the simulation for all the hardware counters tested in this section would have increased considerably. *Output* and *fill* time is not included in the simulation, only *evolve* time with *communication* between processes. It can be seen that for more than 32 processes the efficiency decays considerably for the ICB-ITIC cluster, but not for the Mendieta cluster.

Benchmark results from the LAMMPS web page are also shown in the same figure, on the right panel. They include our results in both ICB-ITIC and Mendieta clusters and other benchmarks from LAMMPS [46] which were executed in a Cray XT 5 cluster (1920 AMD Opteron 2.4GHz processors located in Sandia National Laboratory, SNL) and a Dell T7500 dual hex-core desktop (12 cores Xeon 3.47GHz at SNL). Also shown in Fig. 5 right panel are the results from the Repast HPC [33] code. Both mature codes show behavior which is similar to the behavior of our CA code, including the decay in performance in the ICB-ITIC cluster for more than 32 processes. These results strongly suggest that our parallel domain decomposition approach is reasonably efficient, given that compares well with other mature codes, and that the sudden drop in efficiency beyond 32 cores is hardware-related. Our domain decomposition implementation is similar at the performed in the work of Jelinek et al. [16]. The results of the weak scaling in [16] (executed in the Kraken cluster) obtains ∼80% of parallel efficiency for 144 CPU cores (compared with the parallel time in 12 CPU cores, not the serial time in a single CPU core). We obtain ∼86% parallel efficiency for 64 MPI processes in the Mendieta cluster, when comparing to a single core, and ∼87% parallel efficiency, when comparing to 8 cores.

In order to clarify the origin of the performance drop, which might also be expected in the Piledriver architecture, we carried out tests in the FX-8350 mini-cluster. Two configurations for the FX-8350 mini-cluster can be seen in Fig. 5 for the *mpirun* command. We change the *affinity* to the CPUs of the MPI processes. Processor *affinity* allows the execution of a process or thread in a given CPU core, forcing the process scheduler of the OS to forbid migration of the process to another CPU core, and keeping cache coherence. The default configuration (*v1* in Fig. 5) is to bind each MPI process to a specific CPU core. The MPI processes are executed in CPU cores of the same compute unit, sharing resources between them, even when other CPU cores in other compute unit are free. In the second configuration (*v2* in Fig. 5) we disable the bind to core parameter, therefore the Linux Kernel is in charge of selecting in which cores the MPI processes are going to be executed. In this configuration, the Linux kernel executes the MPI processes using one CPU core per compute unit (when possible). For this reason, the *v2* configuration performs better for less than 12 cores. For instance, executing 8 MPI processes with the *v1* configuration, the 8 processes are executed using only one node of the mini-cluster; with the *v2* configuration, the 8 processes are executed using 2 nodes, with 4 CPU cores without sharing compute units, in each node, given better performance at the cost of "wasting" CPU cores.

The performance difference of the CA in the ICB-ITIC cluster compared with the Mendieta cluster was not expected, at least not within this CA which uses *int* data types because the Bulldozer microarchitecture has two *Integer units* per *compute unit*. In the case of the LAMMPS code, which uses double floating point operations, the decrease could be because each compute unit has only one floating point (FP) unit, and two cores share the same FP unit. When the weak scaling it is executed with 32 processes

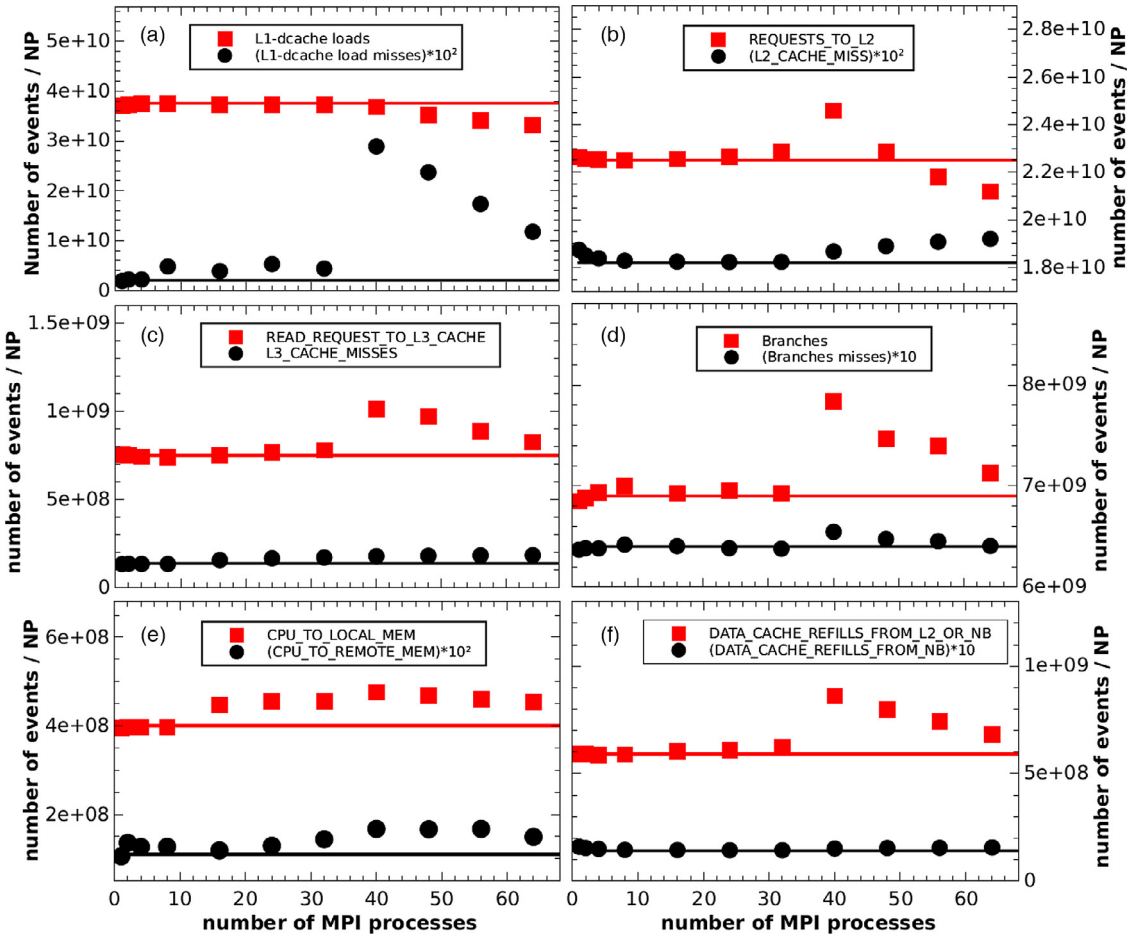

Fig. 6. Weak scaling in ICB-ITIC cluster, with 3000 × 2000 grid, 500 steps. (A) L1 data cache loads with misses, (B) L2 cache loads (event 0x7d) with misses (event 0x7e), (C) L3 cache loads (event 0x4e0) with misses (event 0x4e1), (D) branches with misses, (E) request of CPU to remote and local memory, event 0xe9 with umask (0x98,0xa8), (F) data cache refills from NB (event 0x43) and L2 or NB (event 0x42).

or less, only one integer unit is being used per compute unit and performance begins to degrade when the two integer cores inside each compute unit are being used, with more than 32 MPI processes. The same behavior was reported in the previous subsection, where the strong scaling presented a smaller speedup in the ICB-ITIC cluster (Bulldozer) than in the Mendieta cluster for NP > 16.

In addition to sharing FP units, when two MPI processes are being executed in the same compute unit, they also share the following modules: *L1 instruction cache, instruction fetch, instruction decoder, branch predictor, dispatch unit and L2 data cache*. This suggests that the performance problem could be in the pipeline. To investigate this drop of performance, when more of 32 cores are used in the AMD Opteron 6272 CPU running the CA, we executed weak scaling simulations using perf with hardware counters extracted from the AMD BIOS and Kernel Developers Guide (BKDG) for AMD Family 15h [41].

An event increase of several hardware counters after 32 processes can be seen in Fig. 6A–F. A load miss in the L1 data cache leads to an increase in L2 requests, which can produce a miss in L2 if the data is not present in the cache. An L2 data miss causes a L3 request of data, this behavior can be observed from 32 to 40 processes, the number of L1 data cache misses increases ∼7 times (Fig. 6A). The L2 cache loads (REQUESTS_TO_L2, event 0x7d, umask 0x5f) have a 10% increase from 32 to 40 processes, and the L2 misses (L2_CACHE_MISS, event 0x7e, umask 0x17) scale linearly up to 32 processes, with a 5% increase for 40 processes (Fig. 6B). The L3 related events are shown in Fig. 6C. The event READ_REQUEST_TO_L3_CACHE (L3 loads, event 0x4e0, umask 0xf7) scales linearly up to 32 processes, after that, it increases ∼30% from 32 to 40 processes. The event L3_CACHE_MISSES with code 0x4e1 and umask 0xf7 scales linearly. The branches with misses have a linear scaling up to 32 processes, for 40 processes there is a ∼12% increase in branch references (Fig. 6D), the misses also show a small increase after 32 processes. The event CPU_IO_REQUESTS_TO_MEMORY_IO counts the request of data in a memory in the local processor or data from remote memories, connected to another socket. In Fig. 6E it can be observed this event with code 0xe9 and umask 0x98 (Local CPU to Remote Memory). After 24 processes there is an increase in the number of events, the maximum value is for 40 processes. The same event but with umask 0xa8 (Local CPU to Local Memory) shows an increase of events from 16 processes to 64 processes.

The last two events are shown in Fig. 6F: DATA_CACHE_REFILLS_FROM_L2_OR_NORTHBRIDGE with code 0x42 (umask 0x0f, all sub-events selected) and DATA_CACHE_REFILLS_FROM_NORTHBRIDGE (system memory or another cache) with code 0x43. The refills from the NB scale linearly. The refills from L2 or NB scale linearly up to 32 processes, from 32 to 40 processes the number of events counted increases ∼38%.

The L1 data cache load misses are increasing after 32 processes, which would indicate a loss in performance in the weak scaling for more than 32 processes in the ICB-ITIC cluster. The misses only represent 1% of the cache references, which will not explain by itself the decrease in performance observed in the weak scaling. The L2 and L3 requests also increase after 32 processes, a reasonable behavior due to the L1 data cache misses. One of the reasons for this performance loss could be the size of the L1 data cache in the Opteron 6272 which is 16 KB, the L1 in the Xeon E5-2680 has 32 KB. Another possible problem could be the decoder phase in the pipeline of the Bulldozer microarchitecture, which is shared between the two integer cores inside the compute unit. Further tests are needed to investigate the performance drop in both Weak and Strong scaling in the Bulldozer microarchitecture.

The weak scaling communication time was also tested with the MPI profiler mpiP [47]. For two MPI processes, the communication time for the evolution of the CA was around a 0.5% (without output). With 64 MPI processes, the communication time is less than 6%.

We also tested the CA using the *float* data type for storing the states in the lattice, the behavior of the weak scaling simulations was the same that using the *int* data type, after 32 MPI processes the efficiency decays considerably in the ICB-ITIC cluster, but not in the Mendieta cluster. This is due to the fact that a single floating point unit is present in each compute unit, also the pipeline stages are shared among the running processes in the same compute unit, which will explain the difference in performance compared with the Mendieta cluster.

3.3. Summary of results

In order to understand and improve CA implementations, we first test various serial implementations. Based on this testing, we chose a *One_grid* implementation to perform the evolution of the CA, though commonly two grids are used, one for the current state and one for storing the next state. Different data types, (*char* and *int*), are also tested. Based on our best serial implementation we build a parallel CA implementation using MPI for domain decomposition of large domains.

We use hardware counters to show decrease or improvement in performance for modifications with respect to a *Baseline* serial implementation. We monitor cache references and misses in L1 and LLC, and also branches with misses. For instance, using the *One_grid* approach improves the performance of the baseline code by ∼2.2 ×, because using half the memory gives a more efficient use of all cache types, reducing in most cases cache references and decreasing cache misses in general.

The timing of our parallel *One_grid* implementation for a single process is within 8% of the serial *One_grid* implementation. Weak and strong scaling are carried out, and the performance of our code compares well with the performance of other HPC CA codes [16,20], and also with the performance of two mature HPC codes: the Molecular Dynamics (MD) [32] code LAMMPS [31]; and the Agent-Based-Model (ABM) [19] simulator Repast HPC [33]. MPI calls typically account for less than ∼5% of the total time, reaching ∼20% for the largest grid which uses ∼64 GB RAM, indicating that our domain decomposition approach works reasonably well. Our parallel efficiency is within the obtained by Jelinek et al. [16], it is important to note that the LB simulations performed in [16] are more compute and memory bounded than ours. We note that there is a LB code [10], which performs well in hybrid CPU-GPU clusters.

A decrease in performance was observed for more than 32 cores when executing the CA, LAMMPS and REPAST in the cluster ICB-ITIC (AMD Opteron 6272 processors). This problem was not seen in the Mendieta cluster (Intel Xeon E5 2680 processors), and the fact that the performance drop was observed for three different parallel codes strongly suggest that is related to hardware limitations. It is important to note that the Mendieta cluster has HyperThreading disabled, giving improved performance. One possible reason for this performance loss is the large increase in L1 data cache load misses we observed, but there could be other contributing reasons. Preliminary tests in other AMD Opteron clusters show similar results, but more tests are needed to further clarify the origin of this problem. For instance, performing more tests in the Bulldozer or Piledriver microarchitectures using only two cores in the same compute unit, versus two cores in different compute units could reveal in which phase of the shared (between the two cores) pipeline the execution is stalling. Initial results in the Bulldozer and Piledriver microarchitectures, using one core of each compute unit does show improvements in the performance of the CA. Hardware counters could give us a clear feedback to find the performance problem in these microarchitectures, and then a software modification could be implemented to solve this issue, like software prefetch. The use of inlined PAPI [27] calls could also be used to obtain more detailed info on particular sections of the code, helping to find performance issues and hopefully improve the code for general HPC environments.

4. Conclusions

Parallel implementations of Cellular Automata (CA) scaling well in High Performance Computing (HPC) environments would allow the study of large systems of interest in multiple scenarios, from biology to chemistry and other sciences. The well-known “Game of Life” (GoL) CA was used with a 2D grid, resulting in a memory-bound problem. Several hardware counters are used to quantify performance. Weak and Strong scaling of the CA displays reasonably good scaling, comparable to other HPC applications also using MPI for domain decomposition. The code is written in *C* and uses standard MPI libraries, with GCC and OpenMPI for

compilation; is free and open source (http://goo.gl/9X7tcy), and could be easily converted to work with more complex automata rules.

A CA implementation for modern HPC clusters would need to be flexible enough to run in various architectures, including CPU cores, GPUs, FPGAs, MIPS, etc. Our next step would be to use profiling tools to improve our CA GPU code, and then integrate that code with the MPI implementation shown here, in order to obtain a Multi-GPU and Multi-node implementation of the CA, which would be efficient in a truly hybrid Multi-GPU/Multi-CPU environment, dividing the CA grid (possibly in different size-chunks) and evolving the system using both GPU and CPU strengths. Future software development taking advantage of clusters with hardware accelerators will allow the exploration of novel problems in basic and applied science.

Acknowledgments

E. Millán acknowledges support for his Ph.D. scholarship granted by CONICET. E. Millán and E.M. Bringa thank support from PICT-PRH-0092 and a SeCTyP UNCuyo grant.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.compeleceng.2015.09.015

References

[1] Wolfram S. Cellular automata and complexity: collected papers, 1. Reading: Addison-Wesley; 1994.
[2] Ganguly N, Sikdar BK, Deutsch A, Canright G, Chaudhuri PP. A survey on cellular automata. Dresden University of Technology: Center for High Performance Computing; 2003. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.107.7729.
[3] Pelechano N, Malkawi A. Evacuation simulation models: challenges in modeling high rise building evacuation with cellular automata approaches. Autom Constr 2008;17(4):377–85. http://dx.doi.org/10.1016/j.autcon.2007.06.005.
[4] Silvertown J, Holtier S, Johnson J, Dale P. Cellular automaton models of interspecific competition for space–the effect of pattern on process. J Ecol 1992;80(3):527–33. http://www.jstor.org/stable/2260696.
[5] Topa P, Młocek P. Using shared memory as a cache in cellular automata water flow simulations on gpus. Comput. Sci. 2013;14:3.
[6] Jones DH, Powell A, Bouganis C-S, Cheung PY. A salient region detector for gpu using a cellular automata architecture. In: Neural information processing. Models and applications. Springer; 2010. p. 501–8.
[7] Sahin U, Uguz S, Sahin F. Salt and pepper noise filtering with fuzzy-cellular automata. Comput Electr Eng 2014;40(1):59–69.
[8] Gardner M. Mathematical games: the fantastic combinations of John Conway new solitaire game life. Sci Am 1970;223(4):120–3.
[9] Bandini S, Mauri G, Pavesi G, Simone C. Parallel simulation of reaction-diffusion phenomena in percolation processes. Future Gener Comput Syst 2001;17(6):679–88. http://dx.doi.org/10.1016/s0167-739x(00)00051-0.
[10] Feichtinger C, Habich J, Kstler H, Hager G, Rde U, Wellein G. A flexible patch-based lattice Boltzmann parallelization approach for heterogeneous GPU-CPU clusters. Parallel Comput 2011;37(9):536–49. http://dx.doi.org/10.1016/j.parco.2011.03.005.
[11] Cheng G, Liu L, Jing N, Chen L, Xiong W. General-purpose optimization methods for parallelization of digital terrain analysis based on cellular automata. Comput Geosci 2012;45:57–67. http://dx.doi.org/10.1016/j.cageo.2012.03.009.
[12] Rybacki S, Himmelspach J, Uhrmacher AM. Experiments with single core, multi-core, and GPU based computation of cellular automata. In: First international conference on advances in system simulation, 2009, SIMUL'09. IEEE; 2009. p. 62–7.
[13] Szkoda S, Koza Z, Tykierko M. Accelerating cellular automata simulations using AVX and CUDA, arXiv preprint arXiv:1208.2428.
[14] Tissera PC, Printista AM, Luque E. A hybrid simulation model to test behaviour designs in an emergency evacuation. Procedia Comput Sci 2012;9:266–75.
[15] Chen S, Doolen GD. Lattice Boltzmann method for fluid flows. Annu Rev Fluid Mech 1998;30(1):329–64. http://dx.doi.org/10.1146/annurev.fluid.30.1.329.
[16] Jelinek B, Eshraghi M, Felicelli S, Peters JF. Large-scale parallel lattice Boltzmann cellular automaton model of two-dimensional dendritic growth. Comput Phys Commun 2014;185(3):939–47. http://dx.doi.org/10.1016/j.cpc.2013.09.013.
[17] Pohl T, Deserno F, Thurey N, Rude U, Lammers P, Wellein G, Zeiser T. Performance evaluation of parallel large-scale lattice boltzmann applications on three supercomputing architectures. Proceedings of the ACM/IEEE SC2004 Conference; 2004. http://dx.doi.org/10.1109/sc.2004.37.
[18] Coakley S, Gheorghe M, Holcombe M, Chin S, Worth D, Greenough C. Exploitation of high performance computing in the flame agent-based simulation framework. In: IEEE 14th international conference on high performance computing and communication & 2012 IEEE 9th international conference on embedded software and systems; 2012. http://dx.doi.org/10.1109/hpcc.2012.79.
[19] Macal CM, North MJ. Tutorial on agent-based modelling and simulation. J Simul 2010;4(3):151–62.
[20] Rauch L, Madej L, Spytkowski P, Golab R. Development of the cellular automata framework dedicated for metallic materials microstructure evolution models. Arch Civil Mech Eng 2014. http://dx.doi.org/10.1016/j.acme.2014.06.006.
[21] Oxman G, Weiss S, Be'ery Y. Computational methods for Conway's game of life cellular automaton. J Comput Sci 2014;5(1):24–31. http://dx.doi.org/10.1016/j.jocs.2013.07.005.
[22] Smoller J. Shock waves and reaction-diffusion equations. In: Research supported by the US Air Force and National Science Foundation, 258. New York and Heidelberg: Springer-Verlag; 1983. p. 600.(Grundlehren der Mathematischen Wissenschaften).
[23] Cahn JW, Hilliard JE. Free energy of a nonuniform system. III. Nucleation in a two-component incompressible fluid. J Chem Phys 1959;31(3):688–99.
[24] Farkas D, Caro A, Bringa E, Crowson D. Mechanical response of nanoporous gold. Acta Mater 2013;61(9):3249–56.
[25] Toselli A, Widlund O. Domain decomposition methods: algorithms and theory, 3. Springer; 2005.
[26] Millán EN, Martínez PC, Costa GVG, Piccoli MF, Printista AM, Bederian C, et al. Parallel implementation of a cellular automata in a hybrid CPU/GPU environment. In: De Giusti A, editor. XVIII Congreso Argentino de Ciencias de la Computación, Red de Universidades con Carreras en Informática RedUNCI; 2013. p. 184–93. ISBN 978-987-23963-1-2.
[27] Browne S, Dongarra J, Garner N, London K, Mucci P. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: Supercomputing, ACM/IEEE 2000 Conference. IEEE; 2000. p. 42.
[28] Weaver VM. Linux perf event features and overhead. In: The 2nd international workshop on performance analysis of workload optimized systems, FastPath; 2013. p. 80.
[29] Stallings W. Computer organization and architecture. 5th ed. Prentice Hall; 2000.
[30] Drepper U. What every programmer should know about memory. Red Hat, Inc 2007;11.
[31] Plimpton S. Fast parallel algorithms for short-range molecular dynamics. J Comput Phys 1995;117(1):1–19.
[32] Allen MP, Tildesley DJ. Computer simulation of liquids. Clarendon press Oxford; 1987.
[33] Collier N, North M. Parallel agent-based simulation with repast for high performance computing. Simulation 2013;89(10):1215–35.
[34] Hawick KA, Johnson MG. Bit-packed damaged lattice potts model simulations with cuda and gpus. In: Proceedings of international conference on modelling, simulation and identification; 2011. p. 371–8.

[35] Gibson MJ, Keedwell EC, Savi DA. An investigation of the efficient implementation of cellular automata on multi-core CPU and GPU hardware. J Parallel Distrib Comput http://dx.doi.org/10.1016/j.jpdc.2014.10.011.
[36] Tinetti FG, Martin SM, Frati FE, Méndez M. Optimization and parallelization experiences using hardware performance counters. In: Supercomputing in Mexico: a navigation through science and technology, 4; 2013. p. 157–66.
[37] Malladi RK. Using Intel® VTune performance analyzer events/ratios & optimizing applications.
[38] Treibig J, Hager G, Wellein G. Likwid: Lightweight performance tools. In: Competence in high performance computing 2010. Springer; 2012. p. 165–75.
[39] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Tutorial.
[40] Intel Xeon Processor E5-1600/E5-2600/E5-4600 product families. datasheet, Technical report 326508, Intel.
[41] Devices AM. Bios and kernel developers guide (BKDG) for AMD family 15h models 00h-0fh processors 42301 (3.14). http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/42301_15h_Mod_00h-0Fh_BKDG1.pdf.
[42] Levinthal D. Performance analysis guide for Intel core i7 processor and Intel Xeon 5500 processors 1.0. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf.
[43] Kjolstad FB, Snir M. Ghost cell pattern. In: Proceedings of the 2010 workshop on parallel programming patterns. ACM; 2010. p. 4.
[44] Brunnet LG, Chaté H. Cellular automata on high-dimensional hypercubes. Phys Rev E 2004;69(5):057201.
[45] A. M. D. Advanced Micro Devices, Software optimization guide for AMD family 15h processors. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/03/47414_15h_sw_opt_guide.pdf.
[46] LAMMPS, Lennard–Jones liquid benchmark. http://lammps.sandia.gov/bench.html#lj.
[47] Vetter JS, McCracken MO. Statistical scalability analysis of communication operations in distributed applications. In: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, PPoPP '01. New York, NY, USA: ACM; 2001. p. 123–32. http://dx.doi.org/10.1145/379539.379590.

Emmanuel N. Millán received a BSc. in Software Engineering from the Universidad del Aconcagua, Argentina, in 2010. He is presently pursuing his Ph.D. thesis since 2011 under the supervision of Carlos García Garino, Eduardo Bringa and Maria Fabiana Piccoli. His thesis deals with implementation of parallel problems in hybrid clusters including Graphics Processing Units (GPUs).

Carlos S. Bederian received a MSc. in Computer Science from the Universidad Nacional de Cordoba (UNC), Argentina, in 2008. He is currently working for IFEG-CONICET in Cordoba, and as an Assistant Professor at UNC, specializing in High Performance Computing.

Maria Fabiana Piccoli received her Ph.D. degree from Universidad Nacional de San Luis (UNSL), Argentina, in 2005, and the Graduated in Computer Science degree from UNSL, Argentina, in 1995. She is a full Professor at the UNSL, and director of Departamento de Informática. She is interested in High Performance Computing, including Parallel and Distributed Computing.

Carlos García Garino received his Ph.D. degree from Universidad Politécnica de Cataluña, Spain, in 1993, and the Civil Engineering degree from UBA, Argentina, in 1978. He is a full Professor at the UNCuyo, and director of the ITIC-UNCuyo. He is interested in Computer Networks, Distributed Computing and Computational Mechanics.

Eduardo M. Bringa received his Ph.D. in Physics from UVa (USA) in 2000. He was staff member at Lawrence Livermore National Laboratory (LLNL, USA), and currently holds an Independent Researcher position within CONICET, and also an Associate Professor position at FCEN, UNCuyo. He leads the group on SImulations in Materials, Astrophysics and Physics (SIMAF, https://sites.google.com/site/simafweb/).