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
Available online 17 October 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/© 2015 Elsevier Ltd. All rights reserved.

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 $\sim 10\times$ 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