Para 2010 – State of the Art in Scientific and Parallel Computing – extended abstract no. 34
University of Iceland, Reykjavik, June 6–9 2010
http://vefir.hi.is/para10/extab/para10-paper-34.pdf

Monte Carlo Simulations of Spin Systems on Multi-core Processors

Marco Guidetti*[1], Andrea Maiorano†[2], Filippo Mantovani‡[3], Marcello Pivanti§[1], Sebastiano Fabio Schifano¶[1], and Raffaele Tripiccione‖[1]

[1] *University and INFN Ferrara, I-44100 Ferrara (Italy)*
[2] *University of Roma La Sapienza, I-00100 Roma (Italy)*
[3] *Deutsches Elektronen-Synchrotron DESY, 15738 Zeuthen, Germany*

Abstract We implement Monte Carlo algorithms for the simulation of spin-glass systems and optimize our codes for recent multi-core CPU and GPU architectures. We consider both the *Ising* (binary) and *Heisenberg* (floating-point) spin-glass models. We provide performance figures for the Intel Nehalem and the IBM Cell/BE CPUs and the Nvidia Tesla C1060 GPU; we also draw a comparison with the performance of dedicated machines, such as the Janus system.

Keywords Monte Carlo Simulations, Multi-core Architectures, Spin-Glass Systems

1 Introduction

Spin models are ubiquitous in statistical mechanics, interesting both as descriptions of the properties of several condensed-matter systems and as a paradigm of complexity. The study of the properties of these systems depends critically on numerical Monte Carlo simulations. The computational effort associated to this task is extremely large: a state of the art simulation requires computing resources in the order of tens of thousands of CPU- years. This is made possible by the exploitation of the large amount of parallelism available in the computation combined, for the largest simulations performed in the recent past, with the use of carefully optimized application-driven machines. While a custom solution is still the most performing, the advent of new generation processors (such multi-core CPUs and GPUs) offers new opportunities to tackle these simulations. In this paper we assess the efficiency of these architectures for this computational problem and compare with the performance of dedicated systems.

Spin model are defined in terms of (generalized)-spins, variables defined on the sites i of a discrete hyper-cubic D-dimensional lattice of linear size L. For *discrete* models, spins have values in a finite (usually small) set of numbers; in the simplest case (the Ising model) they take just two values, $s_i = \pm 1, i = 1 \cdots N = L^D$. For continuous models, spins are real variables; in the Heisenberg model, each spin is a 3D vector of unit length $\mathbf{s}_i$ defined at each site of the lattice. A configuration C_k is the set of spins $s_i^{(k)}$ for all lattice sites.

The Monte Carlo dynamics of these systems is governed by simple energy functions:

$$E(C_k) = -\sum J_{ij} \mathbf{s}_i^{(k)} \mathbf{s}_j^{(k)} \tag{1}$$

The sum is taken only on nearest neighbor pairs of sites on the lattice, characterized by interaction parameters J_{ij}. The term $\mathbf{s}_i \mathbf{s}_j$ is a suitably defined scalar product (e.g., the usual scalar product for Heisenberg spins or simply s_i xor s_j for the Ising model. The energy function determines the properties of the system. Spin configurations are distributed according to the Boltzmann distribution

$$P(C_k) \sim e^{-\beta E(C_k)}, \tag{2}$$

corresponding to the probability of the configuration for a system kept at a temperature $T = 1/\beta$.

Monte Carlo algorithms explore the space of configurations, generating random configurations C_k with probability in accordance with 2 (see e.g., [1] for an overview). Statistical averages of observables over the configurations, $O(C_k)$, formally defined as

$$\langle O \rangle = \sum_{C_k} O(C_k) P(C_k), \tag{3}$$

can therefore be estimated as

$$\langle O \rangle \simeq \sum_{C_M} O(C_M) \tag{4}$$

where C_M labels the configurations produced by the Monte Carlo procedure. Monte Carlo algorithms proceed by tentatively replacing the values of the spin at each site of the grid, following appropriate rules (see again [1]) that depend on the contribution of that spin to energy. The key point is that energy depends only on the value of the spin

*Email: guidetti@fe.infn.it
†Email: andrea.maiorano@roma1.infn.it
‡Email: filimanto@gmail.com
§Email: pivanti@fe.infn.it
¶Email: schifano@fe.infn.it
‖Email: tripiccione@fe.infn.it

Algorithm 1 Bit-wise algorithm to update one spin

Require: ρ pseudo-random number
Require: $\psi = \min\{3, \text{int}(-(1/4\beta)\log\rho)\}$ {encoded on two bits}
Require: $\eta = (\text{ not } X_i)$ {encoded on two bits}
1: $c_1 = \psi[0]$ and $\eta[0]$
2: $c_2 = (\psi[1]$ and $\eta[1])$ or $((\psi[1]$ or $\eta[1])$ and $c_1)$
3: $\sigma_i' = \sigma_i$ xor $(c_2$ or not $X_i[2])$

and its nearest neighbors, so a very large amount of parallelism is available: if we consider a checkerboard partition of the grid we can update in parallel all white (or black) sites.

The values of the couplings J_{ij} govern the properties of the system. If all J_{ij} are equal (and positive) these models describe the well-known behavior of a ferromagnetic material. If on the other hands the J_{ij} are randomly extracted (from a bimodal distribution $J_{ij} = \pm 1$ for the Ising model, from a Gaussian distribution with zero mean for the Heisenberg model) we have a true spin-glass model. A ferromagnet at zero temperature will run into one of the two fully magnetized states (all spins aligned in the same direction). At finite temperature it still has two opposite (partially) magnetized states corresponding to (free) energy minima, and can eventually transition between them. For the spin-glass cases (randomly selected J_{ij}) the situation is more complex: due to concurrent couplings the energy landscape becomes rugged, and the time needed to overcome barriers between minima grows exponentially with system size; this is the main reason why simulations are so time consuming.

There are two different opportunities for parallelism in the simulation of these systems: first, we are physically interested in averages of the properties of the system over a large number of independent instantiations of the couplings (we call each such instantiation a *sample*); this implies a large number of independent simulations of systems that do not interact among them. We call this trivial (but useful) parallelism *external*. The second avenue for parallelism (we call it *internal*) exploits the opportunities – described before – of the Monte Carlo dynamics of *each* system. The challenge is then combining both opportunities in the most efficient way for each architecture.

2 Simulation Algorithm

In this section we discuss ways to exploit the parallelism outlined in the previous section. A first approach to internal parallelism is *instruction-level*, using SIMD instructions within each computing-core, and processing several spins in parallel. A further level of internal parallelism exploits *data-level* parallelism, partitioning the whole lattice across the cores, and updating each sub-lattice in parallel. In the following, we show how these options can be best combined for a few target architectures, for both discrete and continuous models.

2.1 Simulation Algorithms for the Ising Spin-Glass

Spin values for binary models can be coded in just one bit, while CPUs operate on long words of k bits (e.g., $k = 32, 64, 128$), so it is useful to combine internal and external parallelism. We proceed by mapping V ($V = 2, \ldots, 16$) binary-valued spins of $w = k/V$ lattice samples on a single CPU-word. We then use SIMD instructions to update in parallel V spins for each of w independent lattices, so we compound an *internal* parallelism of degree V and an *external* parallelism of degree w. We have used this approach first for the IBM/Cell processor (see [2] for details) and then ported it to the Intel architecture, using SSE instructions.

The actual steps of the algorithm (apart from random number generation) are first formally optimized to just a handful of logic operations, described by algorithm 1 (see again [2] for details); we also replace the computation of the logarithm with an access to a look-up table. The main advantages of this scheme are that it exploits the SIMD capabilities of the architecture leveraging on internal and external parallelism and that it does not use conditional statements, badly impacting on performance.

So far, we have discussed how to exploit instruction-level parallelism within a core. We now consider how to exploit multi-core parallelism. The idea is to divide the whole lattice into sub-lattices, and assign each partition to a different core. We split the lattice in C sub-lattices of contiguous planes (C is the number of cores), and we map each sub-lattice of $L \times L \times L/C$ sites onto a different core. Each thread, running on a different core, executes the program defined by a loop in which it first updates all its white spins and then updates all the black ones. White and black spins are stored in data-structures called *half-planes*, each housing $L^2/2$ spins. Each core updates the half-planes of one color performing the steps described by algorithm 2. Each core houses a sub-lattice plus the boundary planes with the adjoining sub-lattices, which have to be updated at end of a sweep of the sub-lattice. Before performing such operation, the cores must be synchronized.

On the Cell processor the details of the implementation vary if the full simulation data-base stays within the local store of the cores of if DMA operations have to be carefully scheduled to move data from/to main memory. For this reason, performances vary significantly as a function of the lattice size (see later). On the Intel Nehalem proces-

Algorithm 2 Program of each thread

1: update the boundaries half-plane (indexes (0) and $((L^3/C)-1)$).
2: **for all** $i \in [1..((L^3/C)-2)]$ **do**
3: update half-planes (i)
4: **end for**
5: exchange half-plane (0) to the *previous core* and half-plane $((L^3/C)-1)$ to the *next core*.

Algorithm 3 program of the host for a GP-GPU simulation

```
1: loop {loop on Monte Carlo steps}
2:    MCupdate(black)
3:    updateBorder()
4:    MCupdate(white)
5:    updateBorder()
6: end loop
```

sor, memory is shared (through a common L3 cache) so the implementation is conceptually simpler. We have implemented and compared the simulation program against:

- different libraries to handle intra-cpu parallelism: openMPI, openMP and pthread.
- shared and distributed memory allocation. In the shared case, the threads share the same data structure allocated by the main program, while in the distributed case each core copies the data items that it needs into a private structure.
- different compilers: we have used both *gcc* and *icc*, the Intel C compiler.

We obtain the best results using the *pthread* library, a *distributed* memory allocation, and the *icc* compiler, (performance gain is of the order of $10-15\%$, compared to the other options). We also find that a synchronization performed by *active waiting* is slightly better than a synchronization by pthread mutex-variables. The program has been written using intrinsic functions to map operations directly to SSE instructions, and can be compiled with a variable number of threads and variable degree of *internal* parallelism . Simulation on GP-GPU use the same update algorithm used for CPUs, but the structure of the computation is slightly different. Following the CUDA terminology, a 3D lattice of side L is divided into a 2D-grid of $(L/2\times L/2)$ sub-lattices. Each sub-lattice is a 3D-grid of $(2\times 2\times L)$ sites. This partition has been chosen for the following main reasons:

- ensure that each block fits on registers and shared memory of the GPU *streaming multiprocessors* (SM), for the most relevant sizes of $L=16\ldots128$,
- generate many blocks to keep active as long as possible each SM and hide memory access latencies,
- generate a large number of *warps* (groups of 8 instructions that can be executed in parallel) per SM to exploit *memory coalescing*, in order to improve the bandwidth between the SM and the memory.

The main simulation-program runs on the host, copies the lattice on the global memory of the GPU – replicating the surface planes – and launches several GPU kernels as described by algorithm 3. The *MCupdate()* kernel is invoked to update the white or black spins. As we need to update $L^3/2$ sites, the kernel runs on a thread-array configured as a 2D-grid of $(L/2\times L/2)$ blocks, where each block is configures as a 3D-array of $(2\times 2\times L)/2$ threads. This configuration allows to have all threads of the block running while the update step is performed. The *updateBorder()* kernel is invoked on a grid of L blocks of L threads; it updates the surface planes of the lattice by performing memory copies in parallel .

Algorithm 4 program of the GP-GPU

```
Require: color = {black, white}
1: load a set of (4 × 4 × L) points
2: apply MC-step to bulk's points
3: save bulk to memory {required to sync blocks}
```

The code executed by the GPU is organized as shown by algorithm 4.

Each block loads a sub-lattice of $(4\times 4\times L)$ sites, including the bulk of spins to update, plus the planes of neighbors necessary for the Monte Carlo update step. As each block has been configures as a 3D-grid of $(2\times 2\times L)/2$ threads, this step is performed by 8 coalesced memory read operation. Each thread then applies the Monte Carlo update step to a single site, and stores the new value to memory. In our implementation we also used the multi-spin coded approach described above. We used one 32-bit word to map one spin plus the value of three coupling variables. This mapping allows to run in parallel up to 8 different simulations.

2.2 Heisenberg Simulation

The Heisenberg model, as outlined in the introduction, requires floating point maths. Accuracy on long simulations requires that double precision be used throughout. As in the *Ising* model, we have exploited instruction parallelism by updating in parallel two non-adjacent spins using SIMD instructions, and data parallelism by dividing the lattice in C sub-lattices, where C is the number of cores, each one housing $(L\times L\times L/C)$ spins. We have developed one implementation for the Intel Nehalem processor and, one for the Nvidia Tesla C1060 GPU.

On the Intel processor we use the *pthread* library to manage parallelism among the cores and the gcc compiler, as it allows to define *vector* variables; operations on such variables are automatically mapped on SSE instructions that update in parallel two non-adjacent spins of the sub-lattice. We also use vector versions of the log and exp functions – heavily used in this code – using intrinsics instructions.

On the GPU we follow the same approach of the *binary* case, but the structure of the thread-array is slightly more complex. As in the binary case, each block updates a sub-lattice of $(2\times 2\times L)$ sites, but due to larger register and shared memory requirements to store the variables of the model, the update procedure is performed by dividing the sub-lattice in blocks of $(2\times 2\times k)$ sites. The value of k that

3D Edwards Anderson Model SUT (ns/spin)				
L	Janus	I-NH	Cell-BE	Tesla C1060
16	0.0016	1.00	0.83	8.48
32	0.0016	0.24	0.40	1.56
48	0.0016	0.32	0.48	1.16
64	0.0016	0.18	0.29	0.72
80	0.0016	0.19	0.82	0.88
96	–	0.32	0.42	0.86
128	–	0.18	0.12	0.64

Table 1: System update time for the 3D Edwards Anderson (binary) model.

3D Heisenberg Model SUT (ns/spin)		
L	I-NH GHz	Tesla C1060
16	55.4 DP	–
32	38.0 DP	34.4 SP / 139.0 DP
48	32.5 DP	29.6 SP / 134.8 DP
64	29.6 DP	31.0 SP / 131.5 DP
80	29.9 DP	28.3 SP / 130.7 DP
96	30.8 DP	29.2 SP / 129.6 DP
128	30.5 DP	28.9 SP / 129.1 DP

Table 2: System update time for the 3D Heisenberg model (SP=single precision, DP=double precision).

uses up all available memory space on the GPU streaming multiprocessor depends on the lattice size L, however $k = 16$ is an acceptable choice for most physically relevant values of L. Our code has been tested on the Nvidia Tesla C1060 GPU.

3 Results, Comparison and Conclusions

In this section we compare performance results for the codes and the machines described above. Our benchmark for the binary case is the *Janus* special-purpose machine, [3], currently the most powerful system available for this class of simulations (Janus, on the other hand, does not support floating-point arithmetics).

For the GPU case, we also quote results for single-precision versions of the code. Indeed, the Tesla C1060 system has rather poor support for double precision, while the recently announced Nvidia *Fermi* [7] architecture does support efficiently double precision: we regard our SP results as educated guesses of what the Fermi architecture may achieve in DP.

We present results using two metrics relevant for physics applications; *system spin update time* (SUT) is the average time needed by the Monte Carlo procedure to update one spin of *one* system, while *global spin update time* (GUT) is SUT divided by the number of replicas that the simulation handles concurrently. SUT is a measure of how well we exploit internal parallelism, while GUT measures both internal and external parallelism. According to the physics program underlying the simulation, either of the metrics (or both) are relevant.

In table 1 and 2 we report our results for the SUT metric. The corresponding GUT values will be presented in the full paper. Some comments are in order:

- Multi-core and GPU architectures have roughly similar performance levels for this class of applications (both binary and floating-point models), even if corresponding peak performances differ by one order of magnitude. We believe that this is due mainly to memory-bandwidth problem and synchronization overheads.

- Widely different multi-core architectures produce performances that hardly differ for more than a factor 4. Performance for the Cell/BE is strongly dependent on lattice size, as the schedule of data transfers to the local store has a strong impact; this effect is less severe for the Nehalem CPU (performance of the latter CPU however drops, in the binary case, as soon as the L3 cache is not large enough for the simulation data-base).

- New architectures reduce by one order of magnitude the gap between special-purpose and commercial systems. However the former machines still have an edge of at least two order of magnitudes.

References

[1] D. P. Landau and K. Binder, A Guide to Monte Carlo Simulations in Statistical Physics, Cambridge University Press (2005)

[2] M. Guidetti, A. Maiorano, F. Mantovani, S. F. Schifano, and R. Tripiccione, Spin Glass Monte Carlo Simulations on the Cell Broadband Engine. *Proceedings of PPAM09 conference, to appear on LNCS Vol. 6067 and 6068*

[3] F. Belletti et al. JANUS: an FPGA-based System for High Performance Scientific Computing. *Computing in Science and Engineering*, **11** (2009), 48-58.

[4] IBM Cell Broadband Engine Architecture, `http://www-128.ibm.com/developerworks/power/\cell/documents.html`

[5] Inside Nehalem: Intel's Future Processor and System, `http://www.realworldtech.com/page.cfm?\ArticleID=RWT040208182719`

[6] NVIDIA's GT200: Inside a Parallel Processor, `http://www.realworldtech.com/page.cfm?\ArticleID=RWT090808195242`

[7] Inside Fermi: Nvidia's HPC Push, `http://www.realworldtech.com/page.cfm?\ArticleID=RWT093009110932`