Journal of Computational Science 66 (2023) 101910

Contents lists available at ScienceDirect

Journal of Computational Science

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

Speedup of the Metropolis protocol via algorithmic optimization

A.E. Macias-Medri [a], G.M. Viswanathan [b], C.E. Fiore [c], M. Koehler [a], M.G.E. da Luz [a,*]

[a] Departamento de Física, Universidade Federal do Paraná, CP 19044, 81531-980, Curitiba-PR, Brazil
[b] Department of Physics and National Institute of Science and Technology of Complex Systems, Universidade Federal do Rio Grande do Norte, 59078-900, Natal-RN, Brazil
[c] Instituto de Física, Universidade de São Paulo, 05315-970, São Paulo-SP, Brazil

ARTICLE INFO

MSC:
00-01
99-00

Keywords:
Metropolis algorithm
Monte Carlo optimization
Pseudo-random number generators
Precalculated Boltzmann factor
Ising model

ABSTRACT

The Metropolis algorithm is widely used in Monte Carlo (MC) simulations in diverse areas of science and technology, especially for problems formulated in terms of lattice models. A common situation is the necessity to perform long sequential processing, e.g., when looking for equilibrium states of distinct physical systems. Hence, even marginal increases in efficiency of the algorithm individual steps can lead to significant reductions in absolute execution runtimes. Usual speedup procedures include hardware updates, parallelization (when possible) and sampling methods. Here we follow a different direction in trying to decrease the full execution times of MC approaches: algorithmic optimization. We show that the algorithms can be improved by implementing relatively few and simple changes in their organization and structure. First, we discuss some refinements for the pseudo-random number generator, addressing the broadly employed Mersenne-Twister algorithm (MT19937-64). Second, we develop a protocol to precalculate the Boltzmann factor, thereby avoiding the high cost of repeatedly calls to this exponential function (indeed, a very recurring step in the standard Metropolis method). To benchmark our proposals we choose the Ising model since it is one of the best known and more extensively studied problems in statistical physics. We consider the mentioned optimizations and different computational elements, like compilers and Hamiltonian variables ranges, testing the efficiency to obtain the system solutions. Our results suggest that the present set of improvement schemes—namely; decreasing the processing time for both, to generate a random number and to implement the one-flip Metropolis step; systematically enforcing optimization for the maximum quantity of algorithm structures accessing random numbers in a code; and considerably reducing the amount of required computations of the MC probabilistic actualization term—might constitute a relevant addition to the existing collection of expediting techniques in MC computational routines.

1. Introduction

Monte Carlo (MC) methods are quite versatile to treat a wide range of problems, either stochastic or deterministic. For instance, Newton's second law and the time-dependent Schrödinger's equation are important examples where direct numerical integration can be used to obtain the proper solutions. For systems with as small number $\mathcal{N}$ of degrees of freedom, simple quadrature procedures can be employed. However, if $\mathcal{N}$ is very large, a different approach becomes necessary. The MC protocol is a fundamental tool exactly for such kind of situation [1–3]. Indeed, in MC integration, the "area under the curve" A (for A a subset of a $\mathcal{N}$-dimensional region R) of a chosen function is calculated by comparing the frequency of random visits to A with that to the whole R. The final estimation can be very close to the exact value, but a long enough number of simulational steps (depending on $\mathcal{N}$ and features of A) may be in order. How these visits to microstates (in the integration example, a coarse graining of R) are implemented and computed are at the heart of MC simulations.

Among different classes of systems, MC methods are particularly important in statistical lattice models [4], providing adequate ways to probe the systems microstates (see Section 3) and to obtain relevant macroscopic quantities by means of averages calculations. Nonetheless, to perform such averages representative equilibrium configurations must be found. Thus, MC requires a lot of iterations (in certain instances $> 10^6$), involving the usage of proper procedures — as the Metropolis algorithm (MA) [5–9] — for sampling (often from a distribution) on a relative big finite set of states. The idea underlying the MA is to generate a great diversity of configurations, with their associated relative weights, through Markov chains of acceptance/rejection probabilities. This requires considerable computational efforts [10,11], whose more expensive aspects are easily identified. Indeed, several numerical operations are concentrated on the calculation of: non-periodic long

* Corresponding author.
E-mail address: luz@fisica.ufpr.br (M.G.E. da Luz).

https://doi.org/10.1016/j.jocs.2022.101910
Received 17 August 2022; Received in revised form 24 October 2022; Accepted 14 November 2022
Available online 19 November 2022

sequences of pseudo-random numbers (PRNs); and the distribution of probabilities, specified through a certain energy functional defined on the system states.

More concretely, on the one hand, (1) algorithms to generate sequences of PRNs with long periods are necessary for many computational applications, in special for those entailing statistical sampling (see, e.g., [12–15]). There are some algorithms of relatively low computational complexity resulting in a fast generation of PRNs (Xorshift [16], Lehmer [17], etc.). However, the sequence period T_{PRN} (i.e., the maximum stretch of "independent" PRNs yielded by the algorithm before systematic repetitions or correlations) is relatively short. For many systems, T_{PRN} should be great enough to allow the dynamics to reach the stationary condition. For instance, in different lattice models MC (with the MA) calculations may demand a period larger than $V \times S \times 10^6$ [18–21], for V the lattice volume and S the number of local possible configurations at any lattice site. Consequently, if the sequence of independent PRNs does not fulfill this requirement, the simulation can generate spurious and even wrong results [22]. An efficient well tested generator of PRNs for MC simulations, bypassing the T_{PRN} length problem, is the Mersenne-twister algorithm (MTA) [23–25].

On the other hand, (2) in MC methods, for each single constituent i of the system — a particle, a spin, a cell, a site, etc. — having the local configuration variable σ_i (assuming a discrete and finite number of values), the macroscopic state is the set $\sigma \equiv \{\sigma_1, \sigma_2, \dots\}$. The transition probabilities for the individual σ_i's depend on a distribution function. If a function of energy — the Hamiltonian — is pre-defined, then the ratio of probabilities between two states (of energy difference ΔE) is given by the Boltzmann factor, namely, $\exp[-\beta \Delta E]$. Here $\beta = 1/(k_B T)$, with T the temperature and k_B the Boltzmann constant. So, depending on the model, a large computational time might be needed to achieve the steady state through probabilistic actualizations in the form $\sigma' \to \sigma'' \to \sigma''' \dots$. A way to speedup the calculations (see, e.g., [26]) is by applying precalculation techniques to the transition probabilities.

Given the aforementioned panorama for the large class of lattice models, this work focus on the MC basic "elements", addressing solely algorithmic advances. Thus, unlike other optimization philosophies (for example, parallelization [27]), we shall examine and verify specific machine code instructions and algorithmic structures to reduce the runtime of the usual MC simulations, i.e., no hardware modifications or new methodologies are been considered. We develop upgrades related to the above (1) and (2) points in the case of lattice models. We discuss how to enhance the MTA as well as how to implement a (local, so very simple) precalculation for the transition probabilities in the MC method. More concretely, our improvements are as it follows.

First, the outputs of more straightforward PRNs generators are often carried out through recurrent procedures as linear-feedback shift register operations or by means of linear congruential relations. As a consequence, one typically gets $T_{PRN} = 2^{64} - 1$ with variables of 64 bits. In contrast, the MTA uses an internal extra process, where operations between elements $A[j]$ $(j = 0, 1, \dots, N_* - 1)$ of an internal vector are computed. Then, the shift register transformations of $A[j]$ are employed to construct the PRNs outputs. This extends T_{PRN} to $2^{19937} - 1$. For each group of N_* generated PRNs the transformations on $A[j]$ are executed once. In this way, the MTA can be viewed as an algorithm with two different stages (more details in the next Section). Since the first stage ($A[j]$ manipulations) is contained in the second one (generating a PRN), some direct optimization schemes during the compilation can be implemented, basically proper organization of memory and of logic gates [28]. With this, a non-negligible increasing of speed for the PRN generator can be achieved.

Second, the previously mentioned difference of energies $\Delta E = E_{\sigma''} - E_{\sigma'}$ of two arbitrary states σ' and σ'' can only take on a finite number of values. Moreover, for local upgrades, i.e., if at each MC step just the site i is randomly altered, only a "local" energy $E^{\{i\}}$, corresponding to a neighborhood $\{i\}$ of i, will change so that $E_{\sigma''} - E_{\sigma'} = E^{\{i\}}_{\sigma''} - E^{\{i\}}_{\sigma'}$. Thus $\{\Delta E\}$ is a relatively small set, essentially of the order of the number of possible values of $E^{\{i\}}$. Therefore, one can exploit this fact and from the Hamiltonian to precalculate all the possible values of $\exp[-\beta \Delta E]$. By means of direct indexation, the probabilities exchanges can then be selected during the simulations without the need to perform any intermediary calculations, with a considerable gain of computational time.

A comprehensive survey of the impact of the above procedures in the efficiency of MC simulations, including distinct optimization options of different compilers, is performed considering the well known Ising model. In special, we calculate important quantities for the problem employing an indirect thermodynamic-like analysis, usually very demanding through MC simulations, but here leading to a concrete speed up due to our algorithm improvements.

2. Upgrading the Mersenne-Twister algorithm

In this section we review the MTA structure and present two simply ways to optimize its performance. We address the 64-bit version in the C language, known as MT19937-64 and originally developed by Matsumoto and Nishimura [23]. Details about the MTA initialization procedures will not be discussed here.

As previously mentioned, the MTA is divided into two stages: to transform all the bits of an internal vector $A[j]$ and to yield output values by means of shift register operations (see functions in Algorithm 1). The function RandInt64OA returns a 64-bit unsigned PRN, which is generated by means of two kinds of bit periodic transformations of $A[j]$ $(j = 0, 1, \cdot, N_* - 1)$. In the first stage (from line 13 to 23), one defines a pivot index j_*, where if $j_* \geq N_*$ two cross loops perform nonlinear mixing of values of $A[i]$ with $A[i + k]$ for $k = M_*$ (if $i < N_* - M_*$) and $k = M_* - N_*$ (otherwise). In the second stage (function ShiftRegister(y)), all bits of $A[j]$ are transformed into each other by means of shift register operations. After this last step the pivot index is increased by 1.

From the function RandInt64OA one identifies four algorithm aspects which can be optimized:

1. The constant vector W_* is defined each time the function is invoked. Thus, as a first possible improvement, we can redefine W_* as a global variable (some compilers optimization procedures already do so, but not all).
2. The counter i used in the **for** loops can be replaced by the pivot index j_* as a way of recycling variables. Note this procedure is possible once j_* is reinitialized after the loops.
3. In order to increase the number of algorithmic structures requiring optimization (ASRO) during the compilation, the keyword "**inline**" should be added to the function. This keyword is very powerful to demand optimization from the compilers [29], especially when general subroutines are built-in into a program-code with a specific application. In our case the MTA into MC method.
4. Each PRN is produced by two transforms of $A[j]$ $(j = 0, 1, \dots, N_* - 1)$ regulated by two different step intervals, one taking place at every single step and the other (when the counter $j_* \geq N_*$) at intervals of N_* steps. Since some compilers optimization processes involve simultaneously data reorganization in the RAM memory and expansions and contractions of loops, these two intervals in the function could cause multiple ASRO with logical conflicts [30,31]. A way to avoid this potential problem is to define a new global vector $R[j]$ $(j = 0, 1, \dots, N_* - 1)$, containing the shift register transformations and accessed whenever $A[j]$ is changed (nonlinear mixtures).

These four upgrades are implemented in a new function, RandInt64I4, shown in the Algorithm 2. For a better quantification of the improvements, some test simulations are performed only with the upgrades 1, 2 and 3, through the function RandInt64I3 (the algorithm for this case is not presented). As for 4, in fact one may be induced to think that including $R[j]$ can lead to a decreasing instead of an

Algorithm 1 Global constants/variables and functions for one single step of the original MTA. The global vector A (of size N_* and nonlinear mixing symmetry M_*) contains values that will be transformed in two intervals of steps. The symbols "$\oplus$", "$\succ$", "$\prec$", "$\wedge$" and "$j{+}{+}$" denote, respectively, the 64 bit operations "`xor`", "bit shift right", "bit shift left", "`and`" and "post-increment". The variable $j_* = 0, 1, \cdots, N_* - 1$ is the pivot index, where $A[j_*]$ is used to compute the output of one PRN. Here "**inline**" is the same instruction than that in the `C` and `C++` languages.

```
1: function inline ShiftRegister(y)
2:     y' ← y ⊕ ((y ≻ 29) ∧ 0x5555555555555555)
3:     y' ← y' ⊕ ((y' ≺ 17) ∧ 0x71D67FFFEDA60000)
4:     y' ← y' ⊕ ((y' ≺ 37) ∧ 0xFFF7EEE000000000)
5:     return y' ⊕ (y' ≻ 43)
6: end function
7: Define: N_* ≡ 312, M_* ≡ 156
8: Allocate: A[0 to N_* − 1]
9: j_* ← N_*
10: function RandInt64OA( )
11:     Define: W_*[0 to 1] ≡ {0, 0xB5026F5AA96619E9}
12:     if j_* ≥ N_* then
13:         for i ← 0 to N_* − M_* − 1 do
14:             x ← (A[i]∧0xFFFFFFFF80000000)∨(A[i+1]∧0x7FFFFFFF)
15:             A[i] ← A[i + M_*] ⊕ (x ≻ 1) ⊕ W_*[x ∧ 1]
16:         end for
17:         for i ← N_* − M_* to N_* − 2 do
18:             x ← (A[i]∧0xFFFFFFFF80000000)∨(A[i+1]∧0x7FFFFFFF)
19:             A[i] ← A[i + M_* − N_*] ⊕ (x ≻ 1) ⊕ W_*[x ∧ 1]
20:         end for
21:         x ← (A[N_*−1]∧0xFFFFFFFF80000000)∨(A[0]∧0x7FFFFFFF)
22:         A[N_* − 1] ← A[M_* − 1] ⊕ (x ≻ 1) ⊕ W_*[x ∧ 1]
23:         j_* ← 0
24:     end if
25:     x ← A[j_*++]
26:     return ShiftRegister(x)
27: end function
```

Algorithm 2 The MTA with the four (items 1–4 in the main text) algorithmic changes. Briefly, the vector W_* is redefined as a global constant, the pivot index j_* is recycled in the loops, the keyword "**inline**" is added to the function header and a new vector $R[j]$ is introduced, assuming the values of ShiftRegister($A[j_*]$) whenever $A[j]$ is computed (so, the output is reproduced only from the $R[j_*]$ values).

```
1: Allocate: R[0 to N_* − 1]
2: Define: W_*[0 to 1] ≡ {0, 0xB5026F5AA96619E9}
3: function inline RandInt64I4( )
4:     if j_* ≥ N_* then
5:         for j_* ← 0 to N_* − M_* − 1 do
6:             x ← (A[j_*] ∧ 0xFFFFFFFF80000000) ∨ (A[j_* + 1] ∧ 0x7FFFFFFF)
7:             A[j_*] ← A[j_* + M_*] ⊕ (x ≻ 1) ⊕ W_*[x ∧ 1]
8:             R[j_*] ←ShiftRegister(A[j_*])
9:         end for
10:        for j_* ← N_* − M_* to N_* − 2 do
11:            x ← (A[j_*] ∧ 0xFFFFFFFF80000000) ∨ (A[j_* + 1] ∧ 0x7FFFFFFF)
12:            A[j_*] ← A[j_* + M_* − N_*] ⊕ (x ≻ 1) ⊕ W_*[x ∧ 1]
13:            R[j_*] ←ShiftRegister(A[j_*])
14:        end for
15:        x ← (A[N_*−1]∧0xFFFFFFFF80000000)∨(A[0]∧0x7FFFFFFF)
16:        A[N_* − 1] ← A[M_* − 1] ⊕ (x ≻ 1) ⊕ W_*[x ∧ 1]
17:        R[N_* − 1] ←ShiftRegister(A[N_* − 1])
18:        j_* ← 0
19:    end if
20:    return R[j_*++]
21: end function
```

increasing in the computational efficiency due to a higher usage of RAM memory. This might be true, but only if no compiler optimization options are enabled or the MTA is implemented 'alone', i.e., not as built-in function for another code (cf., point 3 above). To confirm this latter observation, we have performed some simulations with the MTA just to generate PRNs without any extra more 'involved' task.[1] The assumed compilers and optimization flags are shown in Table 1. The results obtained with the distinct options (i.e., different CPUs and algorithms) are summarized in Table 2.

From the simulations we identify two main trends. First, the `icc` compiler (Intel Free distribution) yields more efficient machine instructions in almost all cases. However, this should not be taken as an universal fact, insomuch the considered algorithms are very simple — on purpose, as already explained — in the examples here (but see next Secs). Further, a small difference in the runtime with i7-4790k CPU and RandInt64I3 function is favorable to the `gcc` compiler in this case.

[1] As far as we know, the minimal number of instructions between calls to the MTA (to generate PRNs) in a code already allowing optimization by compilers is just a single operation. So, we have built and run the simplest possible simulation by means of a sum loop, namely:

for $k \leftarrow 0$ **to** Q **do** $P = P +$ RandInt64XX; with XX equal to OA, I3 or I4.

On the other hand, if no operations between PRNs are included, the compilers notify an unusable function RandInt64XX and no optimization procedures are implemented by them.

Second, with the `gcc` compiler, the best performance is achieved from the RandInt64I3 function (26% better than RandInt64I4). Contrastingly, there is no relevant distinction between RandInt64I3 and RandInt64I4 for `icc`. Thus, as above mentioned, when the algorithms are rather straightforward (notice that in the present illustration they contain only few ASRO to be processed by the compiler), to implement all the fourth upgrades, 1–4, does not lead to a noticeable improvement compared to implement only the first three, 1–3. Indeed, in simple algorithms memory accesses become the essential priority for optimization, and consequently, data increment in RAM irremediably decreases performance [32]. Thence, the real advantage in considering RandInt64I4 is for codes presenting much more ASRO, as typical in rather complex applications (more details below).

We should emphasize that there are no qualitative distinction between the PRNs resulting from RandInt64OA, RandInt64I3 and RandInt64I4 since our improvements do not involve changes in logical or mathematical operations. In fact, we have implemented few simple tests to explicitly check the equivalence between the pseudo-random number stream for I3, I4 and OA. Here we mention just one. We have computed the sum P (described in the footnote 1) of a very large number of PRNs generated from the three algorithms. Considering the same initial conditions presented in the Mersenne Twister 64 bit version website,[2] the calculated P's sums are exactly the same, namely, 5642137241471645936 (details in the footnote 1). This indicates that the improvements I3 and I4 do not change the random number stream from the original algorithm OA.

If specific procedures and/or operations are altered or added to the MTA (e.g., to modify the sequence period), then extra speed gain may be attained. With this aim, we built a direct protocol to generate 32-bit unsigned PRNs from RandInt64XX functions (XX = OA, I3 or I4),

[2] http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/emt64.html.

Table 1
Compilation setup. Two sets of optimization flags are considered for each compiler. The first (`-mG` for the `gcc` compiler or `-mI` for the `icc` compiler) is oriented to mathematical functions, whereas the second (`-fG` and `-fI`) enables "primitive" levels of optimization in loops and RAM memory. Other flags considered (`-O3` and `-Ofast`), common in both compilers, are the highest optimization levels aimed to a single core/thread of CPU. There are many other possible optimization flags for each compiler. However, from many tests they have not led to significant runtime gain, thus they are not discussed here. Note that specialized flags for CPU architectures (`-march` and/or `-mtune`) have not been used. It is important to highlight that the selected flags are intended to algorithmic optimization for a single core/thread only, hence not involving issues like pipelining, parallelization and activity of CPU specialized logic gates.

	gcc version 9.3.0		icc version 14.0.3

`-mG`	`-lm`
	`-ffast-math`
	`-fgraphite-identity`
	`-floop-parallelize-all`
		`-mI`	`-limf`
			`-fimf-precision=low`
			`-ftz`
			`-fma`

`-fG`	`-funroll-all-loops`
	`-fasynchronous-unwind-tables`
	`-fsched2-use-superblocks`
		`-fI`	`-use-intel-optimized-headers`
			`-opt-multi-version-aggressive`
			`-opt-assume-safe-padding`

Table 2
Results for the runtime (in seconds) for the simplest possible sum loop with $Q = 10^9$ steps (see footnote 1). Compiling details are presented in Table 1. The CPUs used in the tests are: Intel Core i7-4790K, Intel Core i7-2600 and Intel Core i7-950. Each value is the mean of five runs.

	gcc	4790K	2600	950	icc	4790K	2600	950
RandInt64OA	*(no flags)*	7.65	11.91	15.41	*(no flags)*	2.11	3.24	4.61
	`-O3`	3.39	4.22	5.44	`-O3`	2.10	3.21	4.57
	`-Ofast`	3.41	4.22	5.42	`-Ofast`	2.09	3.21	4.57
	`-fG`	7.62	11.83	15.41	`-fI`	2.10	3.24	4.61
	`-fG -O3`	3.24	4.12	5.00	`-fI -O3`	2.10	3.21	4.57
	`-fG -Ofast`	3.09	3.89	5.00	`-fI -Ofast`	2.09	3.21	4.57
RandInt64I3	*(no flags)*	5.79	9.34	11.10	*(no flags)*	2.11	3.22	4.57
	`-O3`	2.19	3.47	4.94	`-O3`	2.10	3.21	4.57
	`-Ofast`	2.19	3.47	4.94	`-Ofast`	2.10	3.21	4.57
	`-fG`	5.79	9.34	11.06	`-fI`	2.11	3.22	4.56
	`-fG -O3`	2.06	3.26	4.60	`-fI -O3`	2.09	3.21	4.57
	`-fG -Ofast`	2.07	3.26	4.59	`-fI -Ofast`	2.10	3.20	4.57
RandInt64I4	*(no flags)*	9.89	13.09	17.64	*(no flags)*	2.48	3.81	5.55
	`-O3`	2.86	4.29	5.73	`-O3`	2.46	3.77	5.61
	`-Ofast`	2.81	4.28	5.78	`-Ofast`	2.46	3.77	5.61
	`-fG`	9.86	13.08	17.67	`-fI`	2.47	3.80	5.55
	`-fG -O3`	2.82	4.34	5.43	`-fI -O3`	2.46	3.77	5.61
	`-fG -Ofast`	2.81	4.34	5.44	`-fI -Ofast`	2.46	3.77	5.61

depicted in Algorithm 3. Obviously, such a strategy is justified only if the PRNs reduction from 64 to 32 bits does not introduce appreciable precision problems for the related applications. For instance, for lattice models with volume $V < 2^{32}$ this should not be an issue.

In Algorithm 3, the function RandInt32 returns a unsigned number containing the first (if $\tilde{s}$ = True) or last (if $\tilde{s}$ = False) 32 bits of RandInt64XX. In this way, the 64 bits of a PRN generated by RandInt64XX is divided into two halves, each leading to a 32-bit PRN. Observe that RandInt64XX is called only once, when $\tilde{s}$ = True, for each two calls of RandInt32[3]. This method reduces the period of the PRNs by a factor of 2^{-32} [33], but which is insignificant compared to the MTA period of $2^{19937} - 1$. The PRNs generated from the RandInt32 are tested and compared with those from the RandInt64XX in the Appendix. The results confirm the quality of our pseudo-random numbers of 32 bits.

Analysis of the performance (again, a sum loop **for** $k \to 0$ **to** Q **do** $P = P$+RandInt32) of the Algorithm 3 is displayed in Table 3. In contrast to Table 2, now the `gcc` compiler is more efficient than the `icc` when optimization flags are enabled. Also, notice that RandInt64I3 is still the best option, but this time 15% better than RandInt64I4: in Table 2, the runtime difference between RandInt64I3 and RandInt64I4 is of about 26%. We have pointed out that a great number of ASRO (usually emerging in more complex codes) should increase the efficiency of the RandInt64I4. In fact, RandInt32 incorporates a new and very simple ASRO, thus resulting in some optimization improvement. Such tendency becomes even more clear for the Ising model analyzed in the following Sections. Also, how exactly one can profit from calling once the PRN generator and get two PRNs in the context of MC simulations for lattice models will be illustrated in the next Section.

Finally, for many applications, the precision and periodicity of 32-bit PRNs are more than enough. So, they could be further split (for

Algorithm 3 The function to generate 32-bit unsigned integer PRNs. The global variables $\tilde{s}$ and $\tilde{w}$ are used to determine and then split the first (from 1 to 32) and second (from 33 to 64) parts of a 64-bit PRN. The variable $\tilde{w}$ contains all the bits computed from RandInt64XX. The first half of this 64 bits list is obtained as ($\tilde{w}$ mod MaxInt32), whereas the second as ($\tilde{w}$/MaxInt32). The symbol "¬" means the "`not`" operator.

1: **Define:** MaxInt32≡ 2^{32} ▷ MaxInt64≡ (MaxInt32)^2
2: $\tilde{s} \leftarrow$ True
3: $\tilde{w} \leftarrow 0$
4: **procedure** INLINE RandInt32( )
5: $\tilde{s} \leftarrow \neg\tilde{s}$
6: **if** $\tilde{s}$ **then return** ($\tilde{w}$ mod MaxInt32)
7: $\tilde{w} \leftarrow$ RandInt64XX( ) ▷ XX = OA, I3 or I4.
8: **return** $\tilde{w}$/MaxInt32
9: **end procedure**

[3] One may argue that it would be even faster RandInt32 already to return the two 32-bit PRNs and the MC algorithm to be adapted to use these two numbers in a row before a next call to the function. Of course, this could be implemented if extra efficiency gain would be absolutely mandatory. Here we prefer to maintain our algorithms as general as possible, not addressing dedicated MC implementations.

Table 3
Results for the runtime (in seconds) with Algorithm 3 (32-bit PRN) for a sum loop of $Q = 10^9$ cycles (see footnote 1). The CPUs and optimization flags tested here are the same than those in Table 2.

	gcc	4790K	2600	950	icc	4790K	2600	950
RandInt64OA	*(no flags)*	5.53	8.08	10.79	*(no flags)*	1.69	2.84	3.44
	-O3	2.27	2.78	4.18	-O3	2.33	2.57	3.40
	-Ofast	2.16	2.65	4.17	-Ofast	2.34	2.57	3.40
	-fG	5.53	8.08	10.58	-fI	1.68	2.84	3.44
	-fG -O3	1.99	2.55	3.52	-fI -O3	2.34	2.57	3.40
	-fG -Ofast	1.88	2.55	3.36	-fI -Ofast	2.31	2.57	3.40
RandInt64I3	*(no flags)*	4.54	6.76	8.27	*(no flags)*	1.68	2.83	3.44
	-O3	1.60	2.38	3.31	-O3	2.30	2.70	3.77
	-Ofast	1.49	2.35	3.30	-Ofast	2.30	2.69	3.78
	-fG	4.54	6.77	8.27	-fI	1.68	2.84	3.44
	-fG -O3	1.53	2.28	3.15	-fI -O3	2.30	2.70	3.78
	-fG -Ofast	1.42	2.25	3.15	-fI -Ofast	2.30	2.70	3.78
RandInt64I4	*(no flags)*	6.21	8.90	11.98	*(no flags)*	2.15	2.64	3.78
	-O3	2.10	2.64	3.52	-O3	2.48	3.43	4.73
	-Ofast	1.69	2.66	3.48	-Ofast	2.47	3.43	4.73
	-fG	6.21	8.91	11.92	-fI	2.15	2.64	3.78
	-fG -O3	1.68	2.84	3.37	-fI -O3	2.48	3.43	4.73
	-fG -Ofast	1.68	2.84	3.37	-fI -Ofast	2.47	3.43	4.73

example, to 16-bit), since the period of the MTA is extremely big. This could be easily implemented by repeating the same scheme to our RandInt32 function.

3. Boltzmann factor precalculation method

In the following we address two other aspects that can be algorithmically improved in typical MC simulations: (i) reduction of the number of PRN's required in the MA and (ii) more efficient calculations of the Boltzmann factor. Given that the Ising is a paradigmatic example of a lattice system [34], for concreteness we implement (i) and (ii) considering the Ising model in a square network of volume $V = L^2$. We emphasize, however, that the ideas proposed here could be easily extended to any other lattice model.

The Ising Hamiltonian is given by ($J > 0$ and B an arbitrary real value)

$$\mathcal{H}_S = -J \sum_{\langle i,j \rangle} S_i S_j - B \sum_i S_i, \tag{1}$$

where $S_i = \pm 1$ is the spin value at the lattice site i, $\langle i, j \rangle$ denotes nearest-neighbor sites i and j, and the parameters J (interaction) and B (external field) are constants. The MA is one of the best approaches to numerically analyze the Ising model [34], for instance, usually being more efficient than the Glauber dynamics [35].

The classical MC method using the MA consists of an iterative process. First a site i is randomly chosen. Second, the i-spin can change its state, $S \to S'$, with a probability given by (throughout this work we set the Boltzmann constant $k_B = 1$, then $\beta = 1/(k_B T) = 1/T$)

$$\mathcal{P}_{S \to S'} = \exp[\min\{0, -(\mathcal{H}_{S'} - \mathcal{H}_S)/T\}]. \tag{2}$$

The standard computational procedure for these two stages, the so called one-flip Metropolis scheme, is for convenience illustrated in the Algorithm 4. From it one can readily pinpoint two issues that could be optimized.

a. The function OneFlipSta requires two PRNs, one to randomly choose the spin and the other to calculate the transition probability. For these tasks, the period of the 32 bit PRN from Algorithm 3 can still be considered large. Thus, we develop a set of instructions to further split such PRNs into two parts, or

$$\begin{aligned} &\text{Limit} \leftarrow \text{Real}(\text{MaxInt32})/\text{Real}(L^2), \qquad \text{(Global)} \\ &U \leftarrow \text{RandInt32}(\,), \\ &r \leftarrow U \bmod L^2, \\ &\tilde{r} \leftarrow U/L^2, \end{aligned} \tag{3}$$

Algorithm 4 Standard one-flip Metropolis prescription in the square lattice case. A square lattice containing one spin per node is established by means of a simple array Θ of size L^2. Successive elements along a row (or along a column) differ from each other by 1 in their corresponding array index. The sum of the four neighbor spins in definition SumNei(r) returns the local magnetization around the site r. The function Real(r) transforms an integer into a single/double precision number. For E_i see the main text.

1: **Define:** $L \equiv 128$
2: **Allocate:** $\Theta[0$ **to** $L^2 - 1]$
3: **Define:** SumNei(r) $\equiv \Theta[r + 1 - (r \bmod L = 0) \times L] + \Theta[r - 1 + ((r - 1) \bmod L = 0) \times L] + \Theta[r + L - (r > L^2 - L) \times L^2] + \Theta[r - L + (r < L) \times L^2]$
4: **Define:** RandReal32 ← Real (RandInt32( ))/Real(MaxInt32)
5: **procedure** INLINE OneFlipSta(T, J, B)
6: $r \leftarrow$ Integer(RandReal32 × Real(L^2))
7: $E_i \leftarrow (J \times$ Real(SumNei(r)) $+ B) \times$ Real($\Theta[r]$) ▷ $\Delta E = 2E_i$ for the Ising model
8: **if** $E_i \le 0$ **then**
9: $\Theta[r] \leftarrow -\Theta[r]$
10: **else**
11: **if** $\exp(-2E_i/T) >$ RandReal32 **then** $\Theta[r] \leftarrow -\Theta[r]$
12: **end if**
13: **end procedure**

and substitute RandReal32 by Real$\tilde{r}$/Limit in line 11 of Algorithm 4. As a consequence, the algorithm will invoke RandInt64XX only once each four times the one-flip subroutine is executed. The resulting protocol constitutes the new function OneFlipSpl (details not shown). It is tested for benchmarking below.

b. Another fundamental process which can be optimized in OneFlipSta (Algorithm 4) is to reduce the number of computations of the exponential function in Eq. (2) as well as to avoid comparisons between 0 and $-(\mathcal{H}_{S'} - \mathcal{H}_S)/T$. To do so, consider (for $\sum_{\langle j \rangle_i}$ denoting a sum over the neighbors j of i)

$$E_i = \Big(J \sum_{\langle j \rangle_i} S_j + B \Big) S_i. \tag{4}$$

If the spin S_i is reversed, E_i above changes its signal. Hence, from Eq. (1) and the discussion just before Eq. (2), the system energy variation due to the one flip mechanism is exactly $2\,E_i$. In this way, the second argument of min in Eq. (2) is equal to $-2E_i/T$. But at a given temperature, $-2E_i/T$ can assume at most $2(z+1) = 10$ values depending on B/J (note that for our square

Algorithm 5 Precalculation of the one-flip procedure (microstates update) in the square lattice case. The function OneFlipVec is a reduced and simplified version of the MA, executing just three instructions. Although the parameters T, J and B are not used, they are maintained only to keep the same notation of the previous OneFlipSta and OneFlipSpl functions, thus easing the comparisons. As in RandInt32, note that here a PRN (U) is split in two parts — $(U \bmod L^2)$ and (U/L^2) — increasing the MC simulations speed. This can be employed if the lattice size L^2 does not demand more than 16 bits (indeed, in our case $L^2 = 128^2$, or 14 bits). The node r is randomly chosen by a 14-bit PRN and the matrix elements $D[s][m]$ are compared with a 18-bit PRN (U/L^2).

```
1: Allocate: D[{−1, +1}][−4 to 4 step 2]
2: procedure INLINE AllExp(T, J, B)
3:     for m ← −4 to 4 step 2 do
4:         D[−1][m] ← exp[−2 × (J × m + B) × (−1)/T]
5:         D[+1][m] ← exp[−2 × (J × m + B) × (+1)/T]
6:     end for
7: end procedure
8: Define: Limit←Real(MaxInt32)/Real(L^2)
9: procedure INLINE OneFlipVec(T, J, B)
10:    U ←RandInt32( )
11:    r ← U mod L^2
12:    if D[Θ[r]][SumNei(r)] > Real(U/L^2)/Limit then Θ[r] ← −Θ[r]
13: end procedure
```

lattice $z = 4$ and SumNei$r$ can return only $-4, -2, 0, +2, +4$). Taking all this into consideration, we can add a small matrix D of size $2 \times (z+1)$ to the algorithm, whose elements must be calculated just when T is changed, as shown in Algorithm 5 (function AllExp).

This version of the one-flip procedure (OneFlipVec, Algorithm 5) requires only a single loading of the global matrix $D[s][m]$ ($s = -1, +1$ and $m = -z, -z+2, \ldots, +z$) by the function AllExp. Such restructured fundamental step of MC algorithm does not include any mathematical function and does not need to distinguish between the two cases in Eq. (2) (see the unique instruction in line 12 of Algorithm 5 contrasting with lines 9 and 12 of Algorithm 4). Therefore, it greatly facilitates the optimization process by the compiler.

In Fig. 1 we analyze the runtimes for the Ising model, comparing different combinations of the one-flip Metropolis method, generation functions for the PRNs and compilers. The estimations show that the function OneFlipVec leads to the best performance. We note that smaller times are obtained with the function RandInt64I4 for any OneFlipYyy (Yyy = Sta, Spl or Vec) and compilers. Moreover, the plots for distinct temperatures in Fig. 1 seem to indicate that the runtimes are mainly related to two factors: (a) the probability calculation ($\exp[-2E_i/T]$ or $D[\cdots][\cdots]$) and (b) the one-flip event ($\Theta[r] \leftarrow -\Theta[r]$). Also, taking into account that at least 90% of these simulations are already at the steady state (see next section), the runtime profiles in Fig. 1, as function of T, can be understood by checking how the algorithm accesses (a) and (b) after reaching the steady state.

On the one hand, for high temperatures ($T > 4.5$), state transitions occur with a relatively high rate because $E_i \le 0$ or $\exp[-2E_i/T]$ is close to 1. Actually, resorting to the definitions

$$\gamma \equiv -\frac{B}{J}, \qquad \text{and} \qquad m_i \equiv \sum_{\langle j \rangle_i} S_j = 0, \ldots, z, \tag{5}$$

$E_i > 0$ if (refer to Eq. (4)): (i) $S_i = 1$ and $m_i > \gamma$ or (ii) $S_i = -1$ and $m_i < \gamma$. Thus, we estimate that for $T \approx 4.5$ — i.e., at relatively high temperatures — the system should pass through intermediate states of order–disorder during its evolution towards the steady state. Note that for either (i) or (ii) holding, the average probability to have $E_i > 0$ is around $1/2$, while for other combinations of S_i, γ and m_i, such probability is lower. On the other hand, for low temperatures ($T < 1.5$), at the equilibrium the ground state with minimal energy $\langle \mathcal{H}_S \rangle_0 = -(J\,z + |B|)V$ is characterized by homogeneous phases, which are achieved faster than those for the system at high T's. These phases remain stable in time once $E_i > 0$ and $\exp[-2E_i/T] \approx 0$.

A peculiar behavior is observed when $-2E_i/T \ll 0$. The calling of the exponential function is relatively frequent in this regime, but the runtime is somehow surprisingly short. For instance, the optimization of internal algorithms calculating $\exp[x]$ with $|x| \gg 0$ (from the mathematical libraries of GNU in the Intel distributions) are not enough to cause a significant decreasing in the runtime. But for $|x| \gg 0$, certain processes to reduce the range of x in $f(x)$ involve subroutines rescaling $f(x) = K\,f(x')$, thus decreasing the double/single floating point precision of x' (see, e.g., [36,37]). Given that the numerical operations to determine $f(x')$ require less bits than for $f(x)$, the lowering of the runtime is explained. Further, the relational operator "$>$" is used to compare $\exp[-2E_i/T]$ with the PRN, as seen in line 11 of algorithm 4 and in line 12 of algorithm 5. In this way, although the PRNs $\text{Real}(U/L^2)$ always have the same floating point precision, that for $\exp[-2E_i/T]$ can drastically change during the simulations. But depending on the order of magnitude of the difference between $D[\cdots][\cdots]$ (which encodes $\exp[-2E_i/T]$) and PRN, the number of CPU bit-to-bit comparisons considerably decreases [38], thus reducing the computing time. This is so because large floating point contrasts can be used by compilers to eliminate unnecessary bit-to-bit operations (for details see Ref. [39]). These arguments are supported by the runtimes obtained with the OneFlipVec function (right panel of Fig. 1), which display the same overall profiles with T than the OneFlipSta and OneFlipSpl functions (we recall that $D[\cdots][\cdots]$ is the only variable affected by the temperature).

The proposed method is not restricted to the Ising model. Just as an illustration, let us briefly comment on the Potts model. In general, the dimensions of the key matrix D in the Algorithm 5 will depend on the number of possible energy values. For example, the Hamiltonian for the q Potts [40] reads

$$\mathcal{H}_S = -\tilde{J} \sum_{\langle i,j \rangle} \delta(S_i, S_j) - \tilde{B} \sum_i \delta(S_i, S_*)\,, \tag{6}$$

where $S_i = 0, \ldots, q-1$, $\delta(S_i, S_j)$ is the Kronecker's delta and $\tilde{J}$, $\tilde{B}$ and S_* are constants. So, the energy difference ΔE resulting from the change $S_i \to S_i'$ is

$$-\Delta E(S_i, S_i') = \tilde{J} \sum_{\langle j \rangle} \Big(\delta(S_i', S_j) - \delta(S_i, S_j) \Big) + \tilde{B} \Big(\delta(S_i', S_*) - \delta(S_i, S_*) \Big). \tag{7}$$

All the distinct combinations between $0 \le S_i \le q-1$, $0 \le S_i' \le q-1$ and their neighbors $\langle j \rangle$ determine the possible values which ΔE can assume. In fact, for z neighbors, at most (e.g., depending on the commensurability of $\tilde{B}/\tilde{J}$) we can have a total of $(2\,z+1)\times 3$ possibilities for ΔE — coming from $\tilde{J} \times (-z, \ldots, +z)$ times $\tilde{B} \times (-1, 0, +1)$. Hence, D should be defined as

$$D[a_i][b_i] = \exp[\tilde{J}\,a_i + \tilde{B}\,b_i], \tag{8}$$

where a_i and b_i are auxiliary matrices, to be computed at the beginning of the simulations

$$a_i = a_i[S_i][S_i'][S_{\langle j \rangle}] = \sum_{\langle j \rangle} \Big(\delta(S_i, S_j) - \delta(S_i', S_j) \Big),$$
$$b_i = b_i[S_i][S_i'] = \delta(S_i, S_*) - \delta(S_i', S_*)\,, \tag{9}$$

for $S_{\langle j \rangle}$ denoting the set of neighbor spins to the site i. The matrices a_i and b_i demand an extra RAM memory of $q^{2+z} + q^2$ bytes. However, this is much more advantageous than the usual procedure of calculating the exponential functions, regardless of the optimization

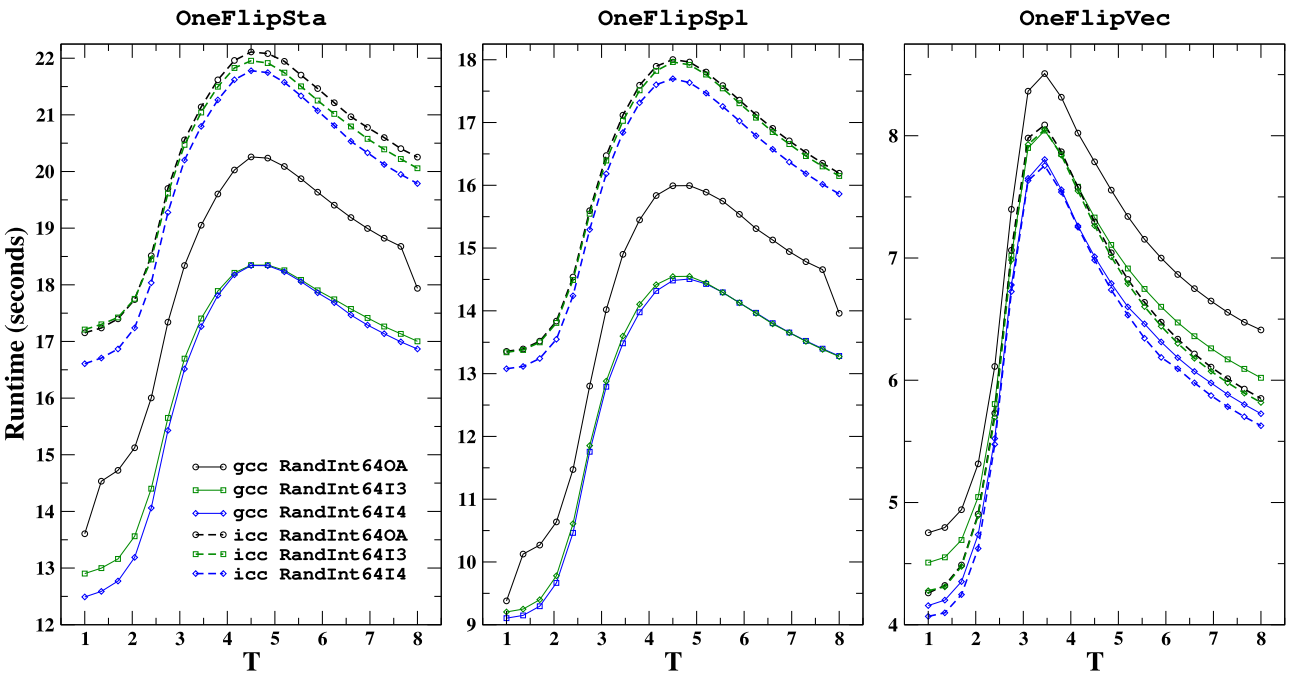

Fig. 1. Runtime (for a CPU Intel i7-4790K) for the Ising model as function of temperature, where $V = 128 \times 128$, $J = 1$ and $B = 0.1$. The plots correspond to averages over 50 simulations, each with 1000 MC steps (much longer than necessary to reach equilibrium). The optimization flags used are those presented in Table 1. In all cases the system starts with randomly oriented spins (a disordered state).

flags. We emphasize that to replace $\exp[\cdot]$ by D does not constitute a parallelization scheme [41,42]. In this way, parallelization procedures plus the $\exp[\cdot] \to D$ protocol should further improve the simulations performance.

As a final observation, it should be noted the computational difficulty to reach the Potts model equilibrium by means of usual MA protocols, specially for large q and lattice sizes (often demanding special methods [43,44]). In particular, there is a critical value for q ($q_c = 4$) above which the probability to find thermodynamically stable states are drastically reduced. In these cases, cluster algorithms [45] and/or tempering schemes [20,46] are often considered to speed up the states space probing. However, the exponential functions are still used in these approaches. The precalculation of $\exp[-\beta\Delta E]$ could easily be extended to such distinct methods. This is a topic we shall explore in a forthcoming contribution.

4. Some thermodynamic simulation tests

As previously mentioned, the results in Fig. 1 with the functions OneFliSpl (Eq. (3)) and OneFlipVec (algorithm 5) have been obtained with 32-bit PRNs, thus speeding up the simulations when compared to the standard function OneFlipSta of 64-bit. But of course, one should guarantee that these changes do not alter the final outcomes, namely, that the computations do lead to the correct thermodynamic quantities for the model.

To verify this we calculate, as function of the number of iterations t (MC steps), the interaction energy (I_t), the magnetization (M_t) and the sum of all auto-correlation functions (G_t^{Φ}), for Φ either I, M, or $\mathcal{H}_S$, defined as

$$I_t = \left\langle \sum_{\langle i,j \rangle} S_i\, S_j \right\rangle, \qquad M_t = \left\langle \sum_i S_i \right\rangle, \qquad G_t^{\Phi} = \frac{1}{t-1} \sum_{s=1}^{t-1} C_{\Phi}(t,s),$$

$$C_{\Phi}(t,s) = \frac{1}{(t-s)\,{\sigma_t}^2} \sum_{\tau=1}^{t-s} \left[\Phi_{(\tau)} - \Phi_t\right]\left[\Phi_{(\tau+s)} - \Phi_t\right], \tag{10}$$

where σ_t is the standard deviation (of t values), $\langle \sum \cdot \rangle$ represents the lattice average and $\Phi_{(t)}$ is the cumulative temporal average up to t, which can be computed as

$$\Phi_{(t)} = \frac{1}{t} \sum_{\tau=1}^{t} \Phi_\tau = \Phi_{(t-1)} + \frac{\Phi_t - \Phi_{(t-1)}}{t}. \tag{11}$$

We observe that usually, for an already well established algorithm, the common practice is to evolve the system beyond the transient time t^* (often estimated from proper trials) and then, for the equilibrium thermodynamic quantities, to perform averages taking $\Phi_{t>t^*}$. Since here we are checking the results for our novel protocols, we prefer to compute the above Φ's as well as the pressure p (see below) in terms of the cumulative mean in Eq. (11). Besides the physical quantities themselves, Eq. (11) also gives a fair idea about the converge of the Φ_t's to their correct values.

The results for G_t^M, $M_{(t)}$ and $I_{(t)}$ as function of t using OneFlipSta and OneFliSpl are displayed in Fig. 2. As can be seen, there are no significant differences between the values of Φ obtained from the OneFlipYyy functions (Yyy = Sta and Spl). This shows that the efficiency in exploring a specific region of the phase space is the same for both kinds of PRNs (64 bit and 32 bit). In other words, the essential elements of the MA are not altered with the present PRNs splitting. Also, we observe from Fig. 2 that the steady states are achieved with relatively few iterations ($t \approx 10$ for $T = 1.0$ and $t \approx 100$ for $T = 4.5$). This corroborates our earlier observation that at least 90% of the total iterations leading to the runtimes in Fig. 1 (with 1000 MC steps) correspond to simulations at the steady state. Finally, prior to reaching the steady states one easily identifies a linear evolution of Φ^M with t. This is mainly due to a competing spin-flip kinetics [47].

Another important check — for the implementation of the OneFliVec function — is to verify if the right convergence is also obtained for distinct values of the Ising model parameters J and B. So, in Fig. 3 we present runtimes and the internal energy density $u \equiv \langle \mathcal{H}_S \rangle = -JI - BM$ for different compilers and values of J and B. We find that the compiler `icc` is more efficient, leading to runtimes around 9–11% shorter. Note this is similar to the results in Fig. 1. For relatively high values of $|J|$, the runtimes diminishes with $|J|$, confirming the positive effect of having great floating point differences between $D[\cdots][\cdots]$ and the PRNs (see the discussion in the paragraph before that of Eq. (6)). For instance,

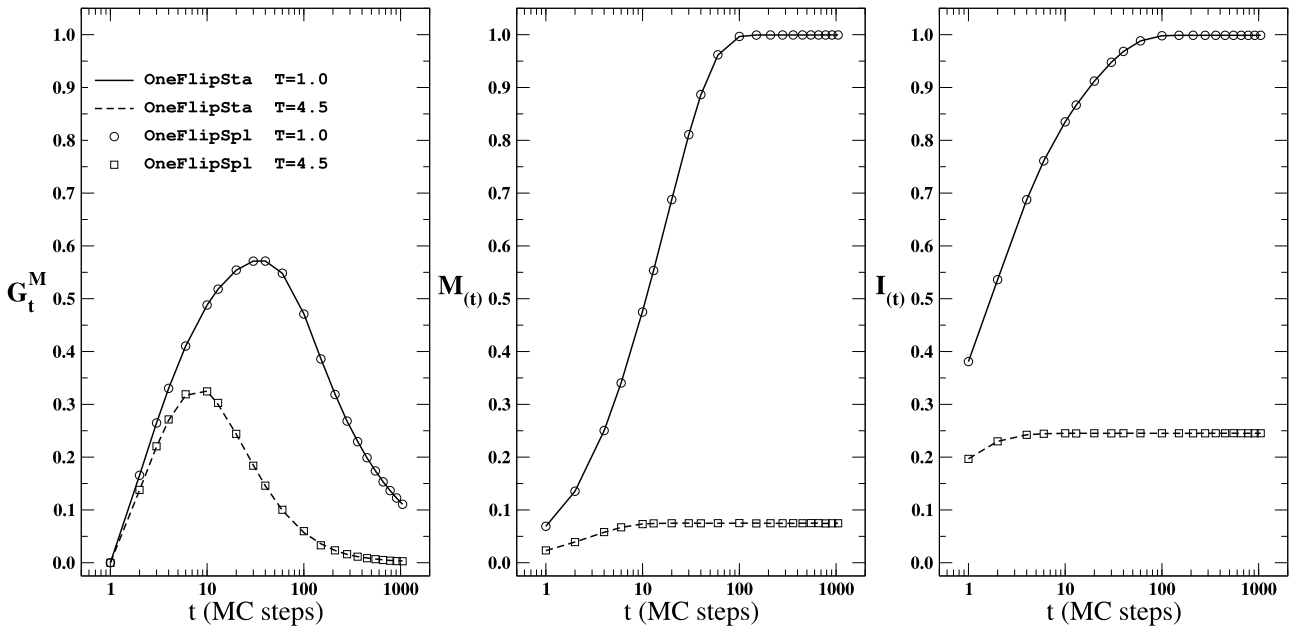

Fig. 2. Evolution with time t (MC steps) of the observables G_t^M, $M_{(t)}$ and $I_{(t)}$ for the Ising model ($J = 1$ and $B = 0.1$) considering the MA for both standard and splitting PRNs. Each curve represents averages over 50 simulation runs in a lattice of volume $V = 128\times128$. In all cases, the system starts with randomly oriented spins, such that $G^M = M = I = 0$ at $t = 0$.

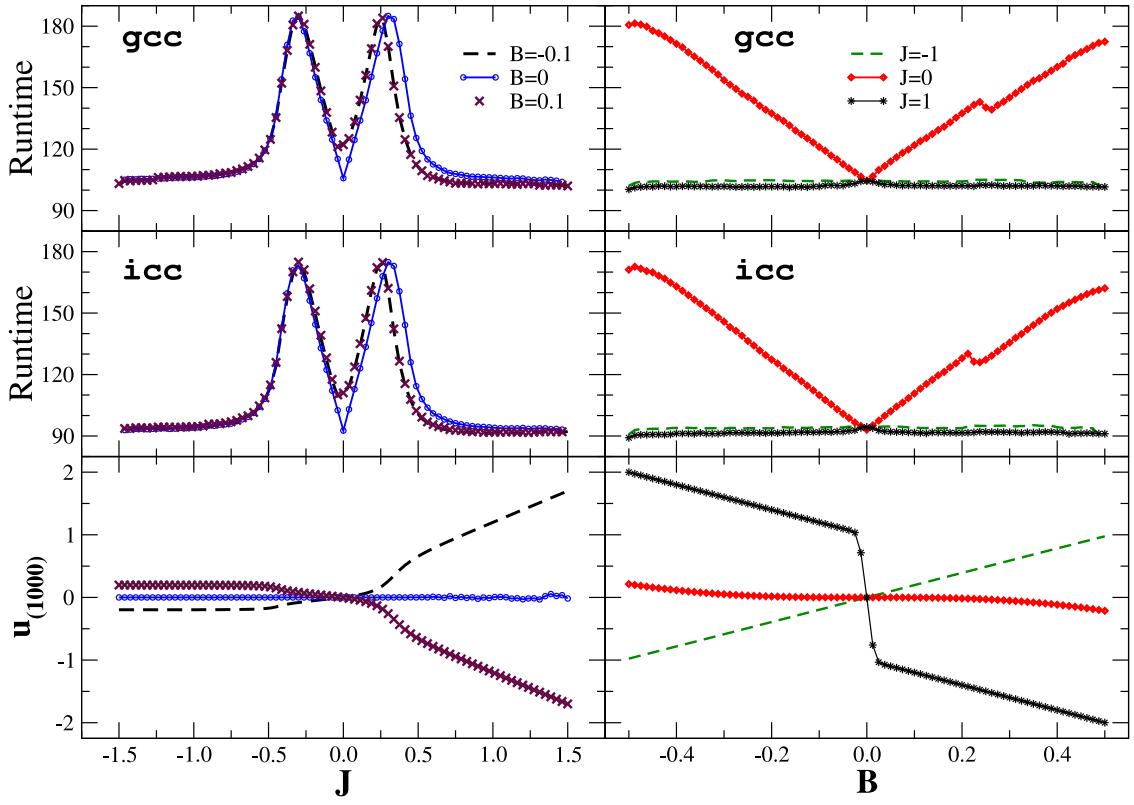

Fig. 3. For $T = 1$, comparisons of the compilers (`gcc` and `icc`) runtimes as well as the behavior of the internal energy density $u_{(1000)}$ as function of J and B. The simulations are performed with the function OneFlipVec, where RandInt64I4 is used in RandInt32 (see algorithm 3). The minimal runtimes correspond to the J (B) value for which all the u curves, for the three distinct B's (J's), do intersect — see footnote 4.

the function AllExp in algorithm 5 (necessary for the precalculation of $\exp[-2E_i/T]$) for low temperatures (e.g., $T = 1$ in Fig. 3) gives $-2E_i/T \approx -2\langle\mathcal{H}_S\rangle_0/(V\,T) = 2(J\,z + |B|)/T$. So, for $|J| \geq 1$ and $B = 0.1$, we have $\exp[-2E_i/T] \sim 10^{\pm3}$, with $\pm$ representing the signal of J. Hence, in both cases the floating point numbers of $D[\cdots][\cdots]$ and Real(U/L^2) differ by a factor of 1000, explaining the outcomes in Fig. 3.

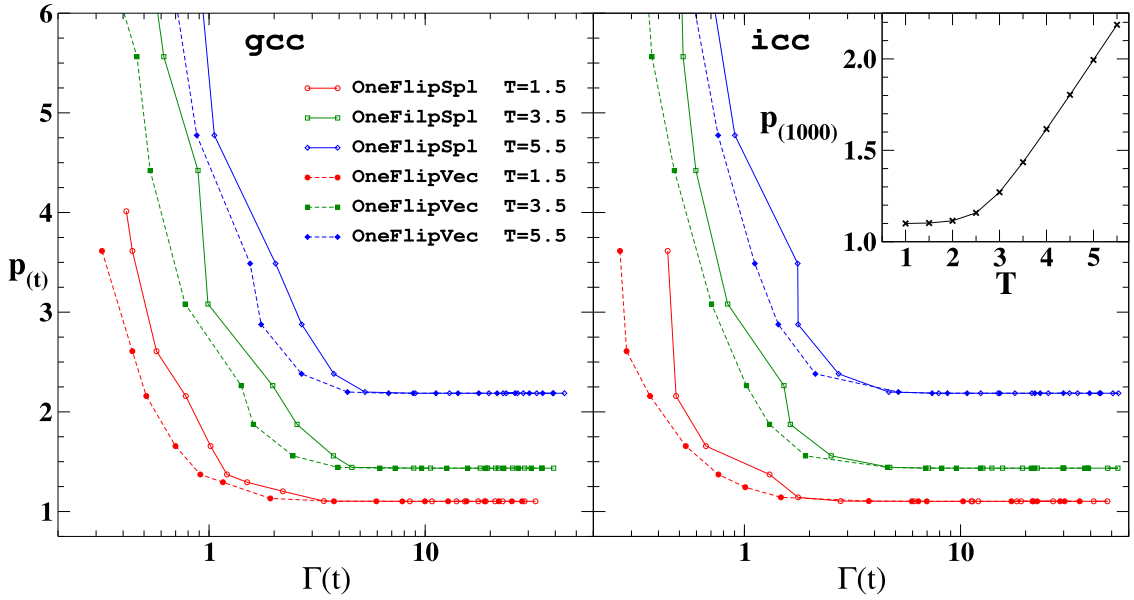

Fig. 4. For distinct T's, the pressure $p_{(t)}(T)$ versus the total, i.e., accumulated runtime $\Gamma(t)$, considering different t MC steps used to calculated the $u_{(t)}(T_n)$'s for the integral in Eq. (16) (see main text). Here $J = 1$, $B = 0.1$, $V = 128 \times 128$ and the reference temperature is $T_0 = 1.0$. Averages are performed over 300 runs with a CPU i7-4790K employing two distinct compilers (`gcc` and `icc`) and the functions OneFlipSpl and OneFlipVec. The assumed t's are 1, 2, 3, 4, 6, 10, 13, 20, 30, 40, 60, 100, 150, 210, 280, 360, 450, 550, 660, 770, 890, 999. The inset shows $p_{(1000)}$ as a function of T.

Inasmuch the $u(J, B)$ curves have been computed at a same low temperature ($T = 1$), crossing lines behavior is observed.[4] For example, a well-known first-order phase transition for the Ising model takes place around $B = 0$. This is identified in the $u(B)$-curves for different values of J, illustrating that the novel algorithm correctly describes the system dynamics. Moreover, a potentially practical useful remark is that for this lattice model (and probably others), phase transitions might be related to local minimums for the runtimes: at an equilibrium state the one-flip events are much less frequent, hence reducing the runtimes.

As a final test, let us discuss a quantity which must be computed indirectly, so in principle a more involved situation. Moreover, to illustrate the efficiency of the present computation approach in a thermodynamic-like analysis for the system, we assume a very simple re-scaling of the free and internal energy densities in terms of a reference temperature T_0 (see, e.g., Refs. [48,49]). In this way, suppose

$$u - \tilde{u} = -\frac{1}{V}\frac{\partial \ln \mathcal{Z}}{\partial \beta}, \qquad \text{with} \qquad \tilde{u} \equiv u(T_0), \tag{12}$$

where $\mathcal{Z}$ is the partition function and T_0 is a low temperature close to the ground state such that the entropy $S \approx 0$. Also, for the free-energy density $\psi(T)$ we write

$$\psi - \tilde{\psi} = -\frac{\ln \mathcal{Z}}{V\beta}, \qquad \text{with} \qquad \tilde{\psi} = \psi(T_0). \tag{13}$$

Eqs. (12) and (13) can be combined to yield

$$-\int_{\beta'}^{\beta''} (u - \tilde{u})\, d\beta = -\beta''(\psi'' - \tilde{\psi}) + \beta'(\psi' - \tilde{\psi}). \tag{14}$$

The pressure is related to the free-energy (grand-canonically) through $\psi = -p$. Then

$$\int_{T'}^{T''} \frac{(u - \tilde{u})}{T^2}\, dT = \frac{p(T'') - p(T_0)}{T''} - \frac{p(T') - p(T_0)}{T'}. \tag{15}$$

For $T' = T_0$ and $T'' = T$, it reads [50]

$$p(T) = -\tilde{u} + T\int_{T_0}^{T} \frac{(u - \tilde{u})}{T^2}\, dT. \tag{16}$$

Now, for $p_{(t)}(T)$ we can proceed as the following. For each t, we calculate $u_{(t)}(T_n)$ at different temperature values $T_n = T_0 + n(T - T_0)/(N - 1)$ (with $n = 0, \ldots, N - 1$). Given T, N is chosen so to lead to a good precision for the numerical integration of Eq. (16). From such scheme, we perform simulations to determine $u_{(t)}(T)$ and the corresponding runtime for t varying from 1 to 999. Finally, setting t we compute both $p_{(t)}$ and the accumulated runtimes $\Gamma(t)$, i.e., the sum of the runtimes to obtain all the $u_{(t)}(T_n)$'s. Fig. 4 presents the results using two distinct compilers. Again, we observe shorter runtimes when the `icc` compiler is used. Also, regarding the efficiency of the functions OneFlipSpl and OneFlipVec, $\Gamma(t)$ is always shorter for the function OneFlipVec.

5. Final remarks and conclusion

In this work we have focused on relatively simple but effective algorithmic improvements to reduce the runtime execution of MC procedures for lattice systems. More specifically, we have addressed and optimized some computational features in the Mersenne-Twister and Metropolis protocols. As a case study we have considered in details the standard Ising model. Nevertheless, it is important to emphasizes that all the innovations proposed here could easily be implemented in similar classes of problems, as briefly discussed for the Potts model in the end of Section 3.

Although lattice models constitute a very relevant class of problems, justifying the present developments, we should observe the following. Taking into consideration the possible types of energy changes for a Hamiltonian H, there are two kinds of Monte Carlo implementations: discrete (DMC) and continuous (CMC) frameworks. The DMC methods are mainly aimed to lattice problems (exactly the situation analyzed here), allowing the precalculation of all probabilities. However, in CMC schemes, Hamiltonians are usually more complex depending on continuous potentials and displaying continuous spectra — refer, e.g., to [51–55]. Of course, in these cases our discretization method (through D) could not be applied. Also for CMC, which still depend on random number generators, the optimization of the PRNs could have a limited

[4] Usually, the first-order phase transition around $B = 0$ is characterized by means of a phase diagram of $M_{(t\gg 0)}$ versus B and T. However, the cross-line method (between two coexisting phases) is also valid for $u(B)$ and $u(J)$ in Fig. 3 once $V\Delta u = T\Delta S$ and $\Delta S \approx 0$ close to the ground state, with S the entropy. Consequently, u is constant in a phase transition.

impact. But this demands extra analysis, outside the scope of our present study.

At this point is useful to summarize and make few extra observations about the main computational developments put forward in the present contribution:

1. Certainly, the precalculation of the Boltzmann factor, given by an exponential function (used in OneFlipSta and OneFlipSpl), will always lead to a performance gain for lattice systems. The key-point is to take advantage of the discrete changes of energy ΔE in the attempts of state updates. Indeed, in some setups we have been able to reduce in 50% the runtimes compared to the traditional method, see Fig. 1. However, this gain can vary according to the compiler (and library) employed. For instance, in our case the `gcc` (`math.h`) has been more efficient than the `icc` (`mathimf.h`).
2. Proper refinements of PRNs generators should diminish the runtimes in MC simulation methods. This fact motivated us to propose four upgrades (Section 2) for the Mersenne-Twister algorithm, which has been incorporated into the functions RandInt64I4 and Rand32Int. However, the concrete gain depends on the features of the compilers optimization flags once the number of "optimization paths" in the solution space for the ASRO increases with the number of conditionals, loops, entanglements, etc [56,57] present in the actual code. For example, the optimization options used for the `icc` compiler are not so efficient for computations employing only the function RandInt64XX into RandInt32 (see Table 3) given that in this case there are just few ASRO. On the other hand, this same compilation configuration — compared to the `gcc` compiler — is the most efficient if the simulations use the OneFlipVec function. We mention that benchmarks contrasting `gcc` and `icc` for other algorithms can be found in [58].

Another important strategy has been to divide a single 64 bit PRN into two 32 bit PRNs (achieved through the substitution of RandInt64XX by RandInt32).[5] But obviously one must be sure that this type of change does not modify the simulated physical quantities, something we have confirmed with the comparisons in Fig. 2. In this regard, the Ising is one of the simplest lattice models available, thus requiring relatively few PRNs ($128^2 \times 10^3$ in Figs. 2–4). Nevertheless, we mention that in a recent work [50] (whose goal was not to discuss algorithmic improvement techniques) some of the authors have used the present PRNs implementation for the much more complex Blume–Emery–Griffiths and Bell–Lavis models, obtaining good results. So, the improvements observed here are not only an artifact of the relative simplicity of the Ising model. We should mention that currently we are testing all these protocols in the simulations of rather involved systems, like some stochastic effects in organic solar cells [59–61], with promising initial findings. The full analysis will be reported in the due course.

We highlight that any study demanding sampling and/or stochastic sorting (associated to Brownian, Poisson, Markov, Branching, Bernoulli, Wiener, and many others processes) and even more sophisticated MC implementations, like the variational [62] and diffusion [63], should also present speedups by means of both the MTA updates and the Boltzmann factor precalculation approach. This also would be the case for replica methods, like the simulated and parallel tempering (for an overview see, e.g., [64]). Indeed, they involve attempts to change the temperature between replicas via an exponential distribution probability, which then could be precalculated exactly as done here.

We hope the present proposals can help future developments in computational physics, specially in the context of statistical lattice models.

[5] It is important to emphasize that although the 64-bit PRNs have been split, the method considered assured that the pseudo-homogeneous distribution of the 32-bit PRNs have not been notably altered from that of the original PRNs.

CRediT authorship contribution statement

A.E. Macias-Medri: Conceptualization, Software, Development, Simulations, Investigation, Methodology, Software, Validation, Visualization, Writing – original draft. **G.M. Viswanathan:** Investigation, Methodology, Visualization, Writing – review & editing. **C.E. Fiore:** Software development, Investigation, Methodology, Software, Writing – review & editing. **M. Koehler:** Project administration, Resources, Methodology, Writing – review & editing. **M.G.E. da Luz:** Conceptualization, Funding acquisition, Methodology, Resources, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

We acknowledge the CNPq grants 302051/2018-0 (Viswanathan), 304532/2019-3 (Luz), 310251/2021-4 (Koeler), 309313/2021-7 (Fiore), as well as financial support from the project "Efficiency in uptake, production and distribution of photovoltaic energy as well as other renewable sources of energy" (Grant No. 88881.311780/2018-00) via CAPES Print-UFPR. This latter project is also acknowledge for a post-doc fellowship (Macias-Medri).

Appendix. Analysis of the 32 bits and 64 bits PRNs

In order to verify the quality of sequences of 32- and 64-bits PRNs, we have considered a total of 15 distinct test indicators (TI) using the standard open source code NIST Statistical Test Suite (NIST-STS) [65], with 10^7 PRNs generated from the algorithms RandInt64OA, RandInt64I4 and RandInt32.

For all the TI's, an important parameter for the analysis is the statistical P-value. Whenever a stream of PRNs give $P > 0.01$, this implies that the sequence can be considered random for most of practical purposes (details in [65]). Also, unless for TI08, TI12 and TI13, for the remaining TI's one can ascribe an unique value for this quantifier.

The analysis for TI01-07, T109-11 and TI14-15 are presented in Table A.4. We first remark that the P-values are always identical for both RandInt64OA and RandInt64I4. This is another indication that the PRNs are not altered by implementing our four improvements. Second, RandInt32 has exact the same P-values than RandInt64XX for TI01-03 and TI06 and almost the same for TI11. For the other TI's in Table A.4 the results differ, but the P-values for RandInt32 are always satisfactory, above $P > 0.01$ (for the smallest, TI07, it is 0.0176).

The test indicators TI08, TI12 and TI13 are depicted in Fig. A.5 for RandInt64OA and RandInt32 (the modified algorithm RandInt64I4 is totally analogous to RandInt64OA, so not shown). Observe that TI12 and TI13 are invariably successful for both algorithms. The TI08, which tries to detect certain periodic overlappings among the considered sequence templates (here 147), has found just one failure ($P < 0.01$) for RandInt64OA (thus 0.68% of the cases) and four for RandInt32 (thus, 2.72% of the cases).

Therefore, we can conclude that the obtained PRNs are very appropriate for the MC simulations.

Table A.4
Twelve test indicators (TI) from the free software NIST-STS random number checker program [65], estimating the quality of 10^7 PRNs having 32- and 64-bits. A pseudo-random number stream is satisfactory if $P > 0.01$.

	RandInt64OA	RandInt64I4	RandInt32
01 Frequency	0.174903	0.174903	0.174903
02 Block Frequency	0.663591	0.663591	0.663591
03 Cumulative Sums	0.318118	0.318118	0.318492
04 Runs	0.126679	0.126679	0.212574
05 Longest Runs of Ones	0.801665	0.801665	0.692375
06 Rank	0.568834	0.568834	0.568834
07 Discrete Fourier Transform	0.454043	0.454043	0.017608
09 Overlapping Template Matchings	0.813638	0.813638	0.466733
10 Universal Statistical	0.435835	0.435835	0.994476
11 Approximate Entropy	0.875363	0.875363	0.865356
14 Serial (lowest)	0.353316	0.353316	0.924530
15 Linear Complexity	0.575134	0.575134	0.738255

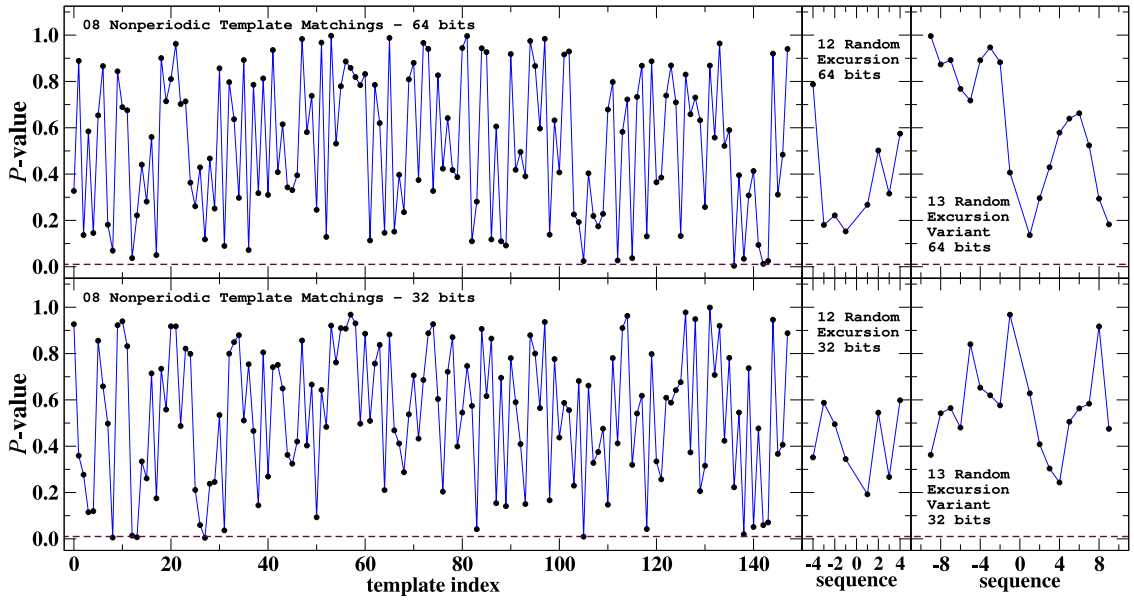

Fig. A.5. Results from the NIST-STS (TI08, TI12 and TI13) for sequences of 10^7 PRNs of 32- and 64-bit (MTA). Very few failures ($P < 0.01$) are identified an only for TI08 (in a total of 147 subtests). For 64-bit it occurs just once (template=111101000) and for 32-bit four times (templates=000010001, 000011011, 000111011, 110011010).

References

[1] W.K. Chan, Theory and Applications of Monte Carlo Simulations, first ed., IntechOpen, London, 2013.

[2] B. Larget, Introduction to Markov chain Monte Carlo methods in molecular evolution, in: R. Nielsen (Ed.), Statistical Methods in Molecular Evolution, Springer, New York, 2005, pp. 45–65, Ch. 3.

[3] D.P. Kroese, T. Brereton, T. Taimre, Z. Botev, Why the Monte Carlo method is so important today? WIREs Comput. Stat. 6 (2014) 386–392.

[4] D.P. Landau, K. Binder, Monte Carlo Simulations in Statistical Physics, fifth ed., Cambridge University Press, Cambridge, 2020.

[5] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines, J. Chem. Phys. 21 (1953) 1087–1092.

[6] I. Beichl, F. Sullivan, The Metropolis algorithm, Comput. Syst. Sci. Eng. 2 (1) (2000) 65–69.

[7] P. Diaconis, L. Saloff-Coste, What do we know about the Metropolis algorithm? J. Comput. Syst. Sci. 57 (1) (1998) 20–36.

[8] B. Ydri, Computational Physics: An Introduction to Monte Carlo Simulations of Matrix Field Theory, World Scientific, Singapore, 2017, pp. 89–104, Ch. 9.

[9] P. Mathé, E. Novak, Simple Monte Carlo and the Metropolis algorithm, J. Complex. 23 (2007) 673–696.

[10] T.M. Yeh, J.J. Sun, Using the Monte Carlo simulation methods in gauge repeatability and reproducibility of measurement system analysis, JART 11 (5) (2013) 780–796.

[11] M.K. Cowles, B.P. Carlin, Markov chain Monte Carlo convergence diagnostics: A comparative review, J. Am. Stat. Assoc. 91 (434) (1996) 883–904.

[12] R.J.N. Baldock, Classical Statistical Mechanics with Nested Sampling (Ph.D. thesis), University of Cambridge, Cambridge, 2017.

[13] L.R. Ernst, R. Valliant, R.J. Casady, Permanent and collocated random number sampling and the coverage of births and deaths, J. Off. Stat. 16 (3) (2000) 211–228.

[14] V. Tirronen, S. Äyrämö, M. Weber, Study on the effects of pseudorandom generation quality on the performance of differential evolution, in: A. Dobnikar, U. Lotrič, B. Šter (Eds.), Adaptive and Natural Computing Algorithms, Springer, Berlin, 2011, pp. 361–370.

[15] D.J. Duffy, Random number generation and distributions, in: D.J. Duffy, J. Kienitz (Eds.), Monte Carlo Frameworks: Building Customisable High-Performance C++ Applications, John Wiley & Sons, Ltd, New Delhi, 2015, pp. 571–599, Ch. 22.

[16] G. Marsaglia, Xorshift RNGs, J. Stat. Softw. 8 (14) (2003) 1–6.

[17] W.H. Payne, J.R. Rabung, T.P. Bogyo, Coding the lehmer pseudo-random number generator, Commun. ACM 12 (2) (1969) 85–86.

[18] H. Suzuki, Monte Carlo simulation of classical spin models with chaotic billiards, Phys. Rev. E 88 (2013) 052144.

[19] J. Anosh, Markov Chain Monte Carlo Methods in Quantum Field Theories: A Modern Primer, Springer, Cham, 2020.

[20] C.E. Fiore, M.G.E. da Luz, Exploting a semi-analytic approach to study first order phase transitions, J. Chem. Phys. 138 (2013) 014105.

[21] F. Faizi, G. Deligiannidis, E. Rosta, Efficient irreversible Monte Carlo samplers, J. Chem. Theory Comput. 16 (2020) 2124–2138.

[22] T.H. Click, G.A. Kaminski, A.B. Liu, Quality of random number generators significantly affects results of Monte Carlo simulations for organic and biological systems, J. Comput. Chem. 32 (3) (2011) 513–524.

[23] M. Matsumoto, T. Nishimura, Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Trans. Model. Comput. Simul. 8 (1) (1998) 3–30.

[24] F. Sepehria, M. Hajivalieia, H. Rajabib, Selection of random number generators in GATE Monte Carlo toolkit, Nucl. Instrum. Methods Phys. Res. A 973 (2020) 164172.

[25] K. Hongo, R. Maezono, K. Miura, Random number generators tested on quantum Monte Carlo simulations, J. Comput. Chem. 31 (2010) 2186–2194.
[26] O.R. Chaparro-Amaro, J. Martínez-Castro, S. Yung-Jun, Vectorization techniques for probability distribution functions using VecCore, JPCS 1525 (2020) 012106.
[27] N.G. Dickson, K. Karimi, F. Hamze, Importance of explicit vectorization for CPU and GPU software performance, J. Comput. Phys. 230 (13) (2011) 5383–5398.
[28] J.M.P. Cardoso, J.G.F. Coutinho, P.C. Diniz, Embedded Computing for High Performance, first ed., Morgan Kaufmann, Boston, 2017, pp. 185–225, Ch. 6.
[29] P.P. Chang, W.W. Hwu, Inline function expansion for compiling c programs, in: R. Wexelblat (Ed.), Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, ACM, New York, 1989, pp. 246–257.
[30] D. Cociorva, J.W. Wilkins, C. Lam, G. Baumgartner, J. Ramanujam, P. Sadayappan, Loop optimization for a class of memory-constrained computations, in: M.M. Furnari, E. Gallopoulos (Eds.), Proceedings of the 15th International Conference on Supercomputing, ACM, New York, 2001, pp. 103–113.
[31] O. Zendra, Memory and compiler optimizations for low-power and -energy, in: R. Ducournau, E. Gagnon, C. Krintz, P. Mulet, J. Vitek, O. Zendra (Eds.), ECOOP 2006: Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, HAL-Inria, Nantes, 2006, p. 8, arXiv:cs/0610028v1.
[32] S. Joannou, R. Raman, An empirical evaluation of extendible arrays, in: P.M. Pardalos, S. Rebennack (Eds.), Experimental Algorithms. SEA 2011, in: Lecture Notes in Computer Science, vol. 6630, Springer, Heidelberg, 2011, pp. 447–458.
[33] J.A. Doornik, Conversion of high-period random numbers to floating point, TOMACS 17 (1) (2007) 3–es.
[34] W. Janke, Monte Carlo simulations in statistical physics – from basic principles to advanced applications, in: Y. Holovatch (Ed.), in: Order, Disorder and Criticality, vol. 3, World Scientific, Singapore, 2012, pp. 93–166, Ch. 3.
[35] W. Janke, H. Christiansen, S. Majumder, Coarsening in the long-range ising model: Metropolis versus glauber criterion, J. Phys.:Conf. Ser. 1163 (2019) 012002.
[36] P.-T.P. Tang, Table-driven implementation of the exponential function in IEEE floating-point arithmetic, ACM Trans. Math. Softw. 15 (2) (1989) 144–157.
[37] J.M. Muller, Range reduction, in: Elementary Functions, Springer, Boston, 1997, pp. 143–162, Ch. 8.
[38] W. Stallings, Computer Organization and Architecture: Designing for Performance, tenth ed., Pearson, London, 2016.
[39] G. Dahlquist, A. Björck, Numerical Methods in Scientific Computing, Vol. 1, SIAM, Philadelphia, 1997.
[40] F.Y. Wu, The Potts model, Rev. Mod. Phys. 54 (1982) 235.
[41] E.E. Ferrero, J.P. De Francesco, N. Wolovick, S.A. Cannas, q-State Potts model metastability study using optimized GPU-based Monte Carlo algorithms, Comp. Phys. Comm. 183 (2012) 1578–1587.
[42] F. Piccini-Cercato, J.C.M. Mombach, G.G.H. Cavalheiro, High performance simulations of the cellular Potts model, in: 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment, IEEE, St.John's-NL, 2006, p. 28.
[43] B.A. Berg, T. Neuhaus, Multicanonical ensemble: A new approach to simulate first-order phase transitions, Phys. Rev. Lett. 68 (9) (1992) 9–12.
[44] F. Wang, D.P. Landua, Efficient, multiple-range random walk algorithm to calculate the density of states, Phys. Rev. Lett. 86.
[45] R.H. Swendsen, J.-S. Wang, Nonuniversal critical dynamics in Monte Carlo simulations, Phys. Rev. Lett. 58 (1987) 86–88.
[46] A. Valentim, M.G.E. da Luz, C.E. Fiore, Determining efficient temperature sets for the simulated tempering method, Comp. Phys. Comm. 185 (2014) 2046–2055.
[47] C. Nuno, First-order phase transition in a 2D random-field ising model with conflicting dynamics, J. Stat. Mech.: Theory Exp. 2009 (02) (2009) P02058.
[48] P. Pfeuty, R.J. Elliott, The ising model with a transverse field. II. Ground state properties, J. Phys. C: Solid State Phys. 4 (15) (1971) 2370–2385.
[49] A.J. Guttmann, I.G. Enting, Series studies of the Potts model. I. The simple cubic ising model, J. Phys. A: Math. Gen. 26 (1993) 807–821.
[50] A.E. Macias-Medri, C.E. Fiore, M.G.E. da Luz, Analyzing and validating simulated tempering implementations at phase transition regimes, Comput. Phys. Commun. 260 (2021) 107256.
[51] P. Fearnhead, J. Bierkens, M. Pollock, G.O. Roberts, Piecewise deterministic Markov processes for continuous-time Monte Carlo, Statist. Sci. 33 (3) (2018) 386–412.
[52] W.L. Jorgensen, J. Tirado-Rives, Molecular modeling of organic and biomolecular systems using BOSS and MCPRO, J. Comput. Chem. 26.
[53] I.C. de Vaca, Y. Qian, J.Z. Vilseck, J. Tirado-Rives, W.L. Jorgensen, Enhanced Monte Carlo methods for modeling proteins including computation of absolute free energies of binding, J. Chem. Theory. Comput. 14 (6) (2018) 3279–3288.
[54] G. Zhou, Mixed Hamiltonian Monte Carlo for mixed discrete and continuous variables, in: H. Larochelle, M. Ranzato, R.T. Hadsell, M.F. Balcan, H. Lin (Eds.), NIPS'20: Proceedings of the 34th International Conference on Neural Information Processing Systems, Curran Associates Inc., Red Hook-NY, 2020, pp. 17094–17104.
[55] M.M. ghahremanpour, J. Tirado-Rives, W.L. Jorfensen, Refinement of the optimized potentials for liquid simulations force field for thermodynamics and dynamics of liquid alkanes, J. Phys. Chem. B 126 (31) (2022) 5896–5907.
[56] R.H. Saavedra, Performance characterization of optimizing compilers, IEEE Trans. Softw. Eng. 21 (7) (1995) 615–628.
[57] K.D. Cooper, S. Devika, T. Linda, Adaptive optimizing compilers for the 21st century, J. Supercomput. 23 (1) (2002) 7–22.
[58] A. Abedalmuhdi, A. Afnan, B. Lakshmy, GCC vs. ICC comparison using PARSEC benchmarks, IJITEE 4 (7) (2014) 76–82.
[59] R.A. Marsh, C. Groves, N.C. Greenham, A microscopic model for the behavior of nanostructured organicphotovoltaic devices, J. Appl. Phys. 101 (8) (2007) 083509.
[60] P.K. Watkins, A.B. Walker, G.L.B. Verschoor, Dynamical Monte Carlo modelling of organic solar cells: The dependence of internal quantum efficiency on morphology, Nano Lett. 5 (9) (2005) 1814–1818.
[61] C. Groves, N.C. Greenham, Monte Carlo simulations of organic photovoltaics, in: D. Beljonne, J. Cornil (Eds.), Multiscale Modelling of Organic and Hybrid Photovoltaics, Springer, Berlin, 2013, pp. 257–278.
[62] B. Rubenstein, Introduction to the variational Monte Carlo method in quantum chemistry and physics, in: J. Wu (Ed.), Variational Methods in Molecular Modeling, Springer, Singapore, 2017, pp. 285–313.
[63] P.J. Reynolds, J. Tobochnik, H. Gould, Diffusion quantum Monte Carlo, Comput. Phys. 4 (1990) 662.
[64] C.E. Fiore, M.G.E. da Luz, Comparing parallel- and simulated-tempering-enhanced sampling algorithms at phase-transitions regimes, Phys. Rev. E 82 (2010) 031104.
[65] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, S. Vo, A statistical test suite for random and pseudorandom number generators for cryptographic applications, Special Publication (NIST SP) 800-22 Rev 1a, National Institute of Standards and Technology, Gaithersburg, MD, USA, 2010.

M. G. E. da Luz: was born in 1968 in Ponta Grossa-PR, Brazil. He achieved a diploma in Physics at the University Federal of Paraná (UFPR) in 1991 and a Doctoral degree in Physics at Unicamp in 1995. He was a post-doc at Harvard University from 1995 to 1997. Since 1998 he has been at the Physics Department at UFPR, becoming a full professor in 2017. For many years he has leading the group of Complexity, Non-Linearity and Disorder in Classical and Quantum Systems at UFPR.