Multithreaded event-chain Monte Carlo with local times

Botao Li, Synge Todo, A. Maggs, Werner Krauth

Version of Record: https://www.sciencedirect.com/science/article/pii/S0010465520303453
Manuscript_214b18ba88bd5b54367e8731e19b5f54

Multithreaded event-chain Monte Carlo with local times

Botao Li[a], Synge Todo[b,c], A. C. Maggs[d], Werner Krauth[a,*]

[a] *Laboratoire de Physique de l'Ecole normale supérieure, ENS, Université PSL, CNRS, Sorbonne Université, Université Paris-Diderot, Sorbonne Paris Cité, Paris, France*
[b] *Department of Physics, University of Tokyo, 113-0033 Tokyo, Japan*
[c] *Institute for Solid State Physics, University of Tokyo, 277-8581 Kashiwa, Japan*
[d] *CNRS UMR7083, ESPCI Paris, PSL Research University, 10 rue Vauquelin, 75005 Paris, France*

Abstract

We present a multithreaded event-chain Monte Carlo algorithm (ECMC) for hard spheres. Threads synchronize at infrequent breakpoints and otherwise scan for local horizon violations. Using a mapping onto absorbing Markov chains, we rigorously prove the correctness of a sequential-consistency implementation for small test suites. On x86 and ARM processors, a C++ (OpenMP) implementation that uses compare-and-swap primitives for data access achieves considerable speed-up with respect to single-threaded code. The generalized birthday problem suggests that for the number of threads scaling as the square root of the number of spheres, the horizon-violation probability remains small for a fixed simulation time. We provide C++ and Python open-source code that reproduces all our results.

Program title: ParaSpheres

Licensing provisions: GNU GPLv3

Programming languages: Python 3, C++, Fortran90

Nature of problem: Multithreaded Event-chain Monte Carlo for hard spheres.

Solution method: Event-driven irreversible Markov-chain Monte Carlo algorithm using local times.

Additional comments: The collection of programs is complete with shell scripts that allow one to reproduce all data, and all the figures of the paper. Change of density and system size is straightforward. The manuscript is accompanied by a frozen copy of the GitHub repository that is made publicly available on GitHub (repository `https://github.com/jellyfysh/ParaSpheres`, commit hash e2aa5b9727fb080ebe65581586c0f6133efa495d).

*Corresponding author, email address: `werner.krauth@ens.fr`

Keywords: Monte Carlo algorithm, irreversible Markov chain, multithreading, event-chain algorithm, sequential consistency model, C++, Python, Fortran90.

1. Introduction

Event-chain Monte Carlo (ECMC) [1, 2] is an event-driven realization of a continuous-time irreversible Markov chain that has found applications in statistical physics [3, 4] and related fields [5]. Initially restricted to hard spheres and to models with piece-wise constant pair potentials [6], ECMC was subsequently extended to continuous potentials, such as spin models and all-atom particle systems with long-range interactions [7, 8]. Potentials need not be pairwise additive [9]. In opposition to standard Monte Carlo methods, such as the Metropolis algorithm [10], ECMC does not evaluate the potential $U(\mathbf{x})$ of a configuration $\mathbf{x}$ (nor any ratio of potentials) in order to sample the Boltzmann distribution $\pi(\mathbf{x}) = \exp[-\beta U(\mathbf{x})]$, with inverse temperature β.

For hard spheres, ECMC is a special case of event-driven molecular dynamics [11, 12]. In molecular dynamics, usually all N spheres have non-zero velocities, and the number of candidate collision events at any time is $\mathcal{O}(N)$. A central scheduler, efficiently implemented through a heap data structure, yields the next collision with computational effort $\mathcal{O}(1)$, and it updates the heap in at most $\mathcal{O}(\log N)$ operations [13, 14]. Event times are global, and the CPU clock advances together with the collision times. The global collision times and the required communications at events complicate multithread implementations [15, 16, 17, 18, 19]. Domain decomposition, another strategy to cope with synchronization, is also problematic [20].

In hard-sphere ECMC, a set $\mathcal{A}_t$ of $k < N$ "active" spheres (all of radius σ) have the same non-zero velocity $\mathbf{v}$ that changes infrequently. All other spheres are "static". At a lifting [21] $l_t = ([i \to j], (\mathbf{x}, \mathbf{x}'), t)$, an active sphere i collides at time t with a target sphere j at contact ($|\mathbf{x}' - \mathbf{x}| = 2\sigma$, a condition that must be adapted for periodic boundary conditions). The lifting l_t connects an in-state (the configuration just before time t, at time t^-) with an out-state (the configuration just after time t, at time t^+):

$$\text{in-state}: \begin{bmatrix} i \in \mathcal{A}_{t^-}, \quad j \notin \mathcal{A}_{t^-} \\ \mathbf{x}_i(t^-) = \mathbf{x} \\ \mathbf{x}_j(t^-) = \mathbf{x}' \\ \mathbf{v}_i(t^-) = \mathbf{v} \\ \mathbf{v}_j(t^-) = 0 \end{bmatrix}; \quad \text{out-state}: \left| \begin{array}{l} i \notin \mathcal{A}_{t^+}, \quad j \in \mathcal{A}_{t^+} \\ \mathbf{x}_i(t^+) = \mathbf{x} \\ \mathbf{x}_j(t^+) = \mathbf{x}' \\ \mathbf{v}_i(t^+) = 0 \\ \mathbf{v}_j(t^+) = \mathbf{v} \end{array} \right|. \quad (1)$$

We consider in this paper two-dimensional spheres in a square box with periodic boundary conditions. In this system, the direction of $\mathbf{v}$ must be changed at certain breakpoints for the algorithm to be irreducible [22]. However, we restrict our attention to ECMC in between two such breakpoints h and h' with, for concreteness, $\mathbf{v} = (v_x, v_y) = (1, 0)$. For a generic "lifted" [21] initial configuration $\{\mathcal{C}_h, \mathcal{A}_h\}$ at h, ECMC is deterministic up to h'. Generically no two

liftings take place at the same time t, so that they can be identified by their time.

Our multithreaded ECMC algorithm propagates $k = |\mathcal{A}|$ active spheres in independent threads, with shared memory. In between h and h', it only uses local-time attributes of each sphere. At a lifting $l_{t_i} = ([i \to j], (\mathbf{x}, \mathbf{x}'), t_i)$, j synchronizes with i (the local time t_j is set equal to t_i). For a sphere i to move, it must not violate certain horizon conditions of nearby spheres j. In the absence of horizon violations between h and h', multithreaded ECMC will be proven equivalent to the global-time process.

The motivation for our work is twofold. First, we strive to speed up current hard-sphere simulations where, typically, $N \sim 1{\times}10^6$. These simulations require weeks or months of run time to decorrelate from the initial configuration [3, 23]. Using a connection to the generalized birthday problem in mathematics, we will argue that such simulations can successfully run with $k \lesssim \sqrt{N}$. Our approach to multithreading thus uses the freedom to tune the number of active spheres. Second, by providing proof of concept for multithreaded ECMC algorithms, we hope to motivate the development of parallel ECMC algorithms for other system where sequential ECMC applies already.

The multithreaded ECMC algorithm is presented in two versions. One implementation uses the sequential-consistency model [24]. Mapped onto an absorbing Markov chain, its correctness is rigorously proven for small test suites. The C++ implementation uses OpenMP to map active chains onto hardware threads, together with atomic primitives [25] for fine-grained control of interactions between threads. Considerable speed-up with respect to a single-threaded version is achieved. The few simultaneously moving spheres ($k \ll N$) avoid communication bottlenecks between threads, even though each hard-sphere lifting involves only little computation.

Subtle aspects of our algorithm surface through the confrontation of the C++ implementation with the sequential-consistency computational model on the same test suites. By reordering single statements in the code, we may for example introduce rare bugs that are not detected during random testing, but are readily exhibited in the rigorous solution, and that illustrate difficulties stemming from possible compiler or processor re-ordering.

Code availability. *Cell-based ECMC for two-dimensional hard spheres is implemented (in Fortran90) as* `CellECMC.f90`*. Our version is slightly modified from two original programs made available in a* `Fortran90/Historic` *directory, written by E. P. Bernard (see Acknowledgements). The code prepares initial configurations. It is used in validation scripts.*

2. Algorithms: from global-time processes to multithreaded ECMC

In this section, we start with the definition of a continuous-time process, Algorithm 1, that is manifestly equivalent to molecular dynamics with the collision rules of eq. (1). Its event-driven version, Algorithm 2, provides the reference set $\mathcal{L}^{\text{ref}}$ of liftings used in our validation scripts (see Section 3). The single-threaded

Algorithm 3 relies on local times. It has correct output if no horizon violation takes place. Its event-driven version, Algorithm 4, yields a practical method that can be implemented and tested. Algorithms 5 and 6 realize multithreaded ECMC, the latter in a highly efficient C++ implementation.

2.1. Continuous processes and ECMC with global time

Algorithm 1 (Continuous process with global time)**.** *At global time $t = h$, an initial lifted configuration $\{\mathcal{C}_h, \mathcal{A}_h\}$ is given ($\mathbf{v}_i = \mathbf{v} = (1, 0)$ $\forall i \in \mathcal{A}_h$ and $\mathbf{v}_i = 0$ $\forall i \notin \mathcal{A}_h$). All spheres i carry local times t_i, with, initially, $t_i(h) = h$ $\forall i$. For active spheres ($i \in \mathcal{A}_t$), $\mathrm{d}t_i/\mathrm{d}t = 1$. At a lifting $l_t = ([i{\to}j], (\mathbf{x}, \mathbf{x}'), t)$ the local time of sphere j is updated as $t_j(t^+) = t$ and, furthermore, $\mathcal{A}_{t^+} = \mathcal{A}_{t^-} \setminus \{i\} \cup \{j\}$. The algorithm stops at global time $t = h'$, and outputs the lifted configuration $\{\mathcal{C}_{h'}, \mathcal{A}_{h'}\}$, and the set $\mathcal{L}_{h'} = \{l_t : h < t < h'\}$ of liftings that have taken place between h and h'.*

Remark 1 (Meaning of local times)**.** *In Alg. 1, the local time $t_i(t)$ is a function of the global time t. It gives the global time at which sphere i was last active (or $t_i(t) = h$ if i was not active for $[h, t]$). Therefore $t_i(t) = t$ $\forall i \in \mathcal{A}_t$ and $t_i(t) < t$ $\forall i \notin \mathcal{A}_t$.*

Remark 2 (Positivity of local-time updates)**.** *In Alg. 1, at any lifting l_t, the update of t_j is positive: $t_j(t^+) - t_j(t^-) > 0$.*

Remark 3 (Time-reversal invariance)**.** *Alg. 1 is deterministic and time-reversal invariant: If an initial lifted configuration $\{\mathcal{C}_h, \mathcal{A}_h\}$ generates the final lifted configuration $\{\mathcal{C}_{h'}, \mathcal{A}_{h'}\}$ with $\mathbf{v}$, then the latter will reproduce the initial configuration with $-\mathbf{v}$. The set $\mathcal{L}$ of liftings is the same in both cases (with exchanged i and j).*

In order to converge towards a given probability distribution, Markov-chain algorithms must satisfy the global-balance condition. It states that the probability flow into a configuration $\mathcal{C}$ (summed over all liftings $\mathcal{A}$) must equal the probability flow out of it [22]. ECMC balances these flows for each lifting individually (for the uniform probability distribution).

Lemma 1. *Alg. 1 satisfies the global-balance condition for any lifted configuration $\{\mathcal{C}, \mathcal{A}\}$. All lifted configurations accessible from a given initial configuration thus have the same statistical weight.*

Proof. The algorithm is equivalent to molecular dynamics that conserves one-dimensional momenta as well as the energy. The claimed property follows for Alg. 1 because it is satisfied by molecular dynamics. The property can be shown directly for a discretized version of Alg. 1 on a rectangular grid aligned with $\mathbf{v}$ with infinitesimal cell size such that each lifted configuration $\{\mathcal{C}, \mathcal{A}\}$ has a unique predecessor. The flow into each lifted configuration equals one. This is equivalent to global balance for the uniform probability distribution. □

The event-driven version of Algorithm 1 is the following:

Algorithm 2 (ECMC with global time)**.** *With input as in Alg. 1, in each iteration $I = 1, 2, \dots$, the next global lifting time is computed as $t_{I+1} = t_I + \min_{i\in\mathcal{A}, j\notin\mathcal{A}} \tau_{ij}$,[1] where τ_{ij} is the time of flight from sphere i to sphere j. At time t_I, local times and positions of active spheres are advanced to $\tilde{t} = \min(t_{I+1}, h')$, and to $\mathbf{x}_i(t_{I+1}) = \mathbf{x}_i(t_I) + (\tilde{t} - t_I)\mathbf{v}$, respectively. If $\tilde{t} = t_{I+1}$ (a lifting $l_{I+1} = (\{i, j\}, \{\mathbf{x}, \mathbf{x}'\}, t_{I+1})$ takes place), the set of active spheres is updated as $\mathcal{A}_{I+1} = \mathcal{A}_I \setminus \{i\} \cup \{j\}$. Otherwise $\tilde{t} = h'$, and the algorithm stops. Output is as in Alg. 1.*

Code availability. *Alg. 2 is implemented in* `GlobalTimeECMC.py` *and invoked in several validation scripts, for which it generates the reference lifting sets $\mathcal{L}^{ref}$.*

2.2. Single-threaded processes and ECMC with local times

Algorithm 3, that we now describe, is a single-threaded emulation of our multithreaded Algorithms 5 and 6. A randomly sampled active chain $\iota \in \{1, 2, \dots\}$ advances (in what corresponds to a thread) for an imposed duration, at most until its local time reaches h'. On thread ι, the active sphere i must remain above the horizons of its neighboring spheres j (see Fig. 1a). The horizon condition is

$$t_i + \tau_{ij} > t_j, \tag{2}$$

where the time of flight is $\tau_{ij} = x_j - x_i + b_{ij}$, with b_{ij} the contact separation parallel to $\mathbf{v}$ between spheres i and j. The horizon condition must be checked for at most three spheres j for a given i because all other spheres are either too far for lifting with i in the direction perpendicular to $\mathbf{v}$ or are prevented from lifting with i by other spheres (see Section 3.1). The algorithm aborts if a horizon violation is encountered. The active chain ι stops if a lifting would be to a sphere j that is itself active. The active chain $\iota + 1$ is then started.

Remark 4 (Double role of horizon condition)**.** *The horizon condition of eq.* (2) *has two roles. First, it is a necessary condition for a lifting of i with j (if it effectively takes place) to produce the required positive local-time update of t_j at the lifting time t (see Remark 2 and Fig. 1a). Second, it is a sufficient non-crossing condition for any sphere k, ensuring that k was not at a previous local time in conflict with i (see Fig. 1b).*

It is for the second role discussed in Remark 4 that the horizon condition is checked for all neighboring spheres j of an active sphere i.

Remark 5 (False alarms from horizon condition)**.** *The horizon condition may lead to false alarms (see Fig. 1b), which could be avoided through the use of the non-crossing condition. The latter is more difficult to check, as it requires the history of past liftings. Our algorithms only implement the horizon condition.*

[1] τ_{ij} is infinite if i cannot lift with j for the given initial configuration $\mathcal{C}$ and velocity $\mathbf{v}$. The presence of an arrow $[i \to j]$ in the directed constraint graph $\mathcal{G}$ indicates that τ_{ij} can be finite (see Section 3.1).

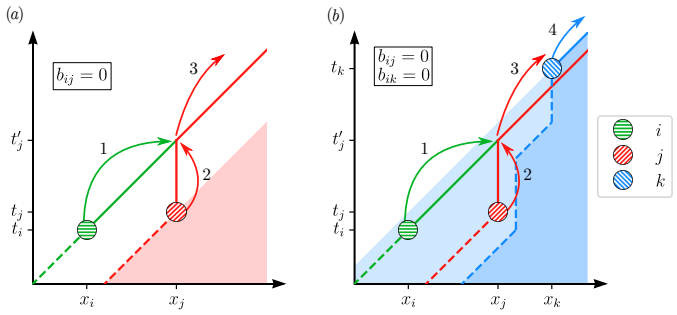

Figure 1: Horizon condition and non-crossing condition in local-time algorithms. (a): Sphere i is above the horizon of sphere j (shaded area), so that at the lifting of i with j, the local-time update of t_j is positive. (b): Sphere i does not lift with k. It does not cross the past trajectory of k, although it violates the horizon condition with k (light shading) (supposing $b_{ik} = 0$). The lifting of i with j could in principle be allowed under the non-crossing condition with k (dark blue shading), supposing $\tau_{jk} = \infty$.

Algorithm 3 (Single-threaded continuous process with local times)**.** *With input as in Alg. 1, active chain $\iota = 1, 2, \ldots$ is initialized (sequentially) with an active sphere i, sampled from $\tilde{\mathcal{A}} = \{i \in \mathcal{A}, t_i \neq h'\}$, and for a local-time interval $\tau_\iota^{\max} = \min(\mathrm{ran}, h' - t_i)$, where* ran *is a positive random number. In active chain ι, the active sphere i moves with velocity $\mathbf{v}$ for $\tau \in [0, \tau_\iota^{\max}]$ and $\mathrm{d}t_i/\mathrm{d}\tau = 1$, if the horizon condition of eq. (2) is satisfied for all spheres j. (In case of a horizon violation, the algorithm aborts.) If a lifting $l_{t_i} = ([i \to j], (\mathbf{x}, \mathbf{x}'), t_i)$ concerns an active sphere j, the active chain ι stops with i at $\mathbf{x}$. Otherwise, the local time of sphere j is updated as $t_j(t_i^+) = t_i$ and $\mathcal{A} = \mathcal{A} \setminus \{i\} \cup \{j\}$, with the active chain ι now moving j. The algorithm terminates if $\tilde{\mathcal{A}} = \emptyset$, that is, if all the active spheres are stalled. Output is as for Alg. 1.*

Remark 6 (Stalled spheres)**.** *"Stalled" spheres (active spheres i with $t_i = h'$) make up the set $\mathcal{A} \setminus \tilde{\mathcal{A}}$. Considering stalled spheres separately simplifies the sampling of $\tilde{\mathcal{A}}$ and the restart from h' for the next leg of the ECMC run.*

Lemma 2. *If Alg. 3 terminates without a horizon violation, its output is identical to that of Alg. 1.*

Proof. We consider the final lifted configuration $\{\mathcal{C}_{h'}, \mathcal{A}_{h'}\}$ of a run that has terminated without a horizon violation, and that has preserved a log of all local-time updates. The termination condition is $\tilde{\mathcal{A}} = \emptyset$, so that all active spheres are stalled with local time h'. We further consider the final lifting $l_{h''}$ in $\mathcal{L}_{h'}$ (so that $t < h'' \; \forall \; l_t \in \mathcal{L}_{h''}$). Local times of static spheres satisfy $t_i \leq h'' \; \forall i \notin \mathcal{A}_{h'}$. When backtracking, using Alg. 1 with $-\mathbf{v}$, from h' to h''^+, no lifting takes place among active spheres (see Fig. 2a). The area swept out by the active spheres cannot overlap with a static sphere j because it must have $t_j < h''$ (local times are smaller than the last lifting) and, on the other hand, $t_j > h''$, because of eq. (2) (see Fig. 2b). The lifting $l_{h''} = ([i \to j], (\mathbf{x}, \mathbf{x}'), h'')$ can now be undone.

(From $j \in \mathcal{A}_{h'}$ and $i \notin \mathcal{A}_{h'}$, we obtain $\mathcal{A}_{h''} = \mathcal{A}_{h'} \setminus \{j\} \cup \{i\}$. The updated local time $t_j(h''^-)$ can be reconstructed from the log. It is smaller than h''. The lifting is then itself eliminated: $\mathcal{L}_{h''} = \mathcal{L}_{h'} \setminus \{l_{h''}\}$.) All active spheres at h''^- now have local time h''. Similary, all liftings can be undone, effectively running Alg. 2 with $-\mathbf{v}$ from h' to h. As Alg. 1 is time-inversion invariant, the local times at its liftings are the same as those of Alg. 3. □

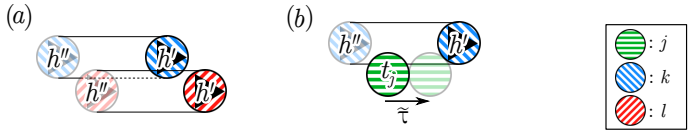

Figure 2: Backtrack using Alg. 1 from the final configuration of Alg. 3. (a): Active spheres k and l do not lift among each other. (b): A static sphere j crossing the trajectory of active sphere k. This crossing is impossible because of the horizon condition ($\tilde{\tau} < h' - h''$ leading to $t_j > h''$, in contradiction with the condition $t_j < h''$).

For concreteness, in the following event-driven formulation of Algorithm 3, the local-time interval $\tau_\iota^{\max}$ of an active chain ι is chosen equal to the time of flight towards the next lifting.

Algorithm 4 (Single-threaded ECMC with local times)**.** *With input as in Alg. 1, for each (sequential) active chain $\iota = 1, 2, \ldots$, an active sphere i is sampled from $\tilde{\mathcal{A}} = \{i \in \mathcal{A}, t_i \neq h'\}$. The horizon conditions of eq. (2) are checked for all*[2] *spheres j that can have finite time of flight τ_{ij}. The algorithm aborts if a violation occurs. Otherwise, i is moved forward to $\min(t_i + \tau_{ij}, h')$, and the local time of i and j are updated to that time. The active chain stops if j is an active sphere or if the local time equals h'. Otherwise, the move corresponds to a lifting $l_{t_i+\tau_{ij}} = ([i \to j], (\mathbf{x}, \mathbf{x}'), t_i + \tau_{ij})$ and $\mathcal{A} = \mathcal{A} \setminus \{i\} \cup \{j\}$, with the active chain now moving j. The algorithm terminates if $\tilde{\mathcal{A}} = \emptyset$. Output is as for Alg. 1.*

Code availability. *Alg. 4 is implemented in* `SingleThreadLocalTimeECMC.py` *and tested in the* `PValidateECMC.sh` *script.*

Remark 7 (Partial validation)**.** *In Section 3.2, a variant of Alg. 4 is used to validate part of a run, even if it does not terminate correctly. When a sphere i detects a horizon violation, its time t_i is recorded. At h', the set $\mathcal{L}_{t^*}$ of all liftings up to the earliest horizon violation, at t^*, agrees with the corresponding partial list of liftings for Alg. 1.*

2.3. Multithreaded ECMC (sequential-consistency model)

Algorithm 5, the subject of the present section, is a model shared-memory ECMC on k threads, that is, on as many threads as there are active spheres. The

[2] at most three spheres j can have finite τ_{ij} for any i, see Section 3.1

algorithm adopts the sequential-consistency model [24]. We rigorously prove its correctness for small test suites by mapping the multithreading stage of this algorithm to an absorbing Markov chain. The algorithm allows us to show that certain seemingly innocuous modifications of Algorithm 6 (the C++ implementation) contain bugs that are too rare to be detected by routine testing.

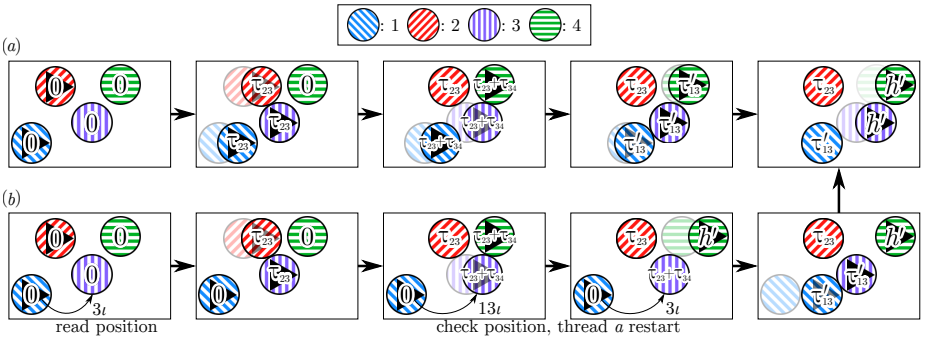

Figure 3: Alg. 5, as applied to the `SequentialC4` test suite. (a): Reference set $\mathcal{L}^{\mathrm{ref}}$ from Alg. 2. (b): Run of Alg. 5 involving a "lock-less" lock rejection on the position of sphere 3.

The algorithm has three stages. In the (sequential) initialization stage, it inputs a lifted initial configuration and maps each active sphere to a thread. This is followed by the multithreading stage, where each active chain progresses independently, checking the horizon conditions in its local environment. The algorithm concludes with the (sequential) output stage.

At each step of the multithreading stage of Algorithm 5, a switch randomly selects one of the k statements (one for each thread $a, b, \dots$) contained in a buffer as $\{\mathrm{next}_a, \mathrm{next}_b, \dots\}$. The selected statement is executed on the corresponding thread, and then the buffer is updated. The random sequence of statements mimics the absence of thread synchronization except at breakpoints. All threads possess an absorbing **wait** statement. When it is reached throughout, the algorithm progresses to the output stage, followed by successful termination. The program aborts when a thread detects a horizon violation. For our test suites, we prove by explicit construction that each state is connected to at least one of the absorbing states, but we lack a general proof of validity for arbitrary configurations and general N.

In Algorithm 5, each sphere has three attributes, namely a tag, a local time and a position. The sphere's tag indicates whether it is active on a thread ι, stalled, or static. All threads have read/write access to the attributes of all spheres. A state of the Markov chain is constituted by the spheres with their attributes, some local variables and the buffer content.

Algorithm 5 (Multithreaded ECMC (sequential-consistency model))**.** *At breakpoint $h = 0$, a lifted initial configuration $\{\mathcal{C}_h, \mathcal{A}_h\}$ is input (see Fig. 3 for the example with four spheres). All local times are set to $h = 0$, all tags are put to static, except for the active spheres, whose tags correspond to their thread*

ι. *The buffer is set to* $\{1_1, \ldots, 1_\iota, \ldots, 1_k\}$. *A random switch selects one buffer element. The corresponding statement is executed on its thread, and the buffer is replenished. The following provides pseudo-code for the multithreading stage* (i_ι *is the active sphere,* j_ι *the target sphere, and* $distance_\iota$ *the difference between* h' *and the local time, all on thread* ι*):*

```
1ι   τ_ι ← distance_ι; j_ι ← i_ι; x_ι ← ∞
2ι   for j̃ in {1, 2, ..., n} \ i_ι :
3ι     x_j̃ ← j̃.x
4ι     τ_ij̃ ← x_j̃ − i_ι.x − b_ij̃
5ι     if i_ι.t+τ_ij̃<j̃.t : abort
6ι     if τ_ij̃ < τ_ι :
7ι       j_ι ← j̃
8ι       x_ι ← x_j̃
9ι       τ_ι ← τ_ij̃
10ι  j_ι.tag.CAS(static, ι)
11ι  if j_ι.tag = ι :
12ι    if τ_ι < distance_ι :
13ι      if x_ι = j_ι.x :
14ι        j_ι.t ← i_ι.t + τ_ι
15ι        i_ι.t ← i_ι.t + τ_ι
16ι        i_ι.x ← i_ι.x + τ_ι
17ι        i_ι.tag ← static
18ι        distance_ι ← distance_ι − τ_ι
19ι        i_ι ← j_ι
         else :
20ι        j_ι.tag ← static
           goto 1
       else :
21ι      i_ι.t ← i_ι.t + τ_ι
22ι      i_ι.x ← i_ι.x + τ_ι
23ι      distance_ι ← 0
24ι      i_ι.tag ← stalled
     else : goto 1
25ι  if distance_ι > 0 : goto 1
26ι  wait
```

When all k *threads have reached their* **wait** *statements, the algorithm proceeds to its output stage. Output is as for Alg. 1.*

Code availability. `SequentialMultiThreadECMC.py` *implements Alg. 5. It also constructs all states connected to the initial state and traces them to the absorbing states. It is called by* `SequentialC4.sh` *and* `SequentialC5.sh`

Remark 8 (Illustration of pseudocode)**.** *The multithreading stage of Alg. 5 corresponds to* k *identical programs running independently. In the sequential-consistency model, the space of programming statements is thus* k*-dimensional*

(one sequence $(1\iota, \ldots, 26\iota)$ *per thread), and each displacement in this space proceeds along a randomly chosen coordinate axis. As an example, if for a buffer* $\{next_1, \ldots, next_\iota = 20\iota, \ldots, next_k\}$ *the switch selects thread* ι*, then the tag of target particle* j_ι *is set to "static", and the buffer is updated to* $\{next_1, \ldots, next_\iota = 1\iota, \ldots, next_k\}$*. The thread* ι *will thus be restarted at its next selection.*

The compare-and-swap (CAS) statement in 10ι of Algorithm 5 amounts to a single-line **if**. It is equivalent to: "**if** $j_\iota.tag = static$ **:** $j_\iota.tag = \iota$" (if j is static, then it is set to active on thread ι (see Remark 9 for a discussion).

We prove correctness of Algorithm 5, for the `SequentialC4` test suite with $N = 2$ and $k = 2$ (see Fig. 3), that we later extend to the `SequentialC5` test suite with $N = 5$.

Lemma 3. *If Alg. 5 terminates without a horizon violation, its output (for the* `SequentialC4` *test suite) is identical to that of Alg. 1.*

Proof. In the `SequentialC4` test suite with threads "a" and "b", we suppose that the switch samples a and b with equal probabilities. The two-thread stage of Alg. 5 then consists in a finite Markov chain with 3670 states S_n that are accessible from the initial state. The **abort** state has no buffer content. All other 3669 states comprise the buffer $\{\text{next}_a, \text{next}_b\}$, the sphere objects (the spheres and their attributes: tag, local time, position), and some thread-specific local variables. One iteration of the Markov chain (selection of next_a or next_b, execution of the corresponding statement, buffer update) realizes the transition from S_n to a state S_m with probability $T_{nm} = 1/2$. The 3670×3670 transition matrix $T = (T_{nm})$ has unit diagonal elements for the **abort**, and for the unique **terminate** state with buffer $\{26a, 26b\}$, which are both absorbing states of the Markov chain. Furthermore, we can show explicitly that all 3670 states have a finite probability to reach an absorbing state in a finite number of steps. This proves that the Markov chain is absorbing. For an absorbing Markov chain, all states that are not absorbing are transient, and they die out at large times. The algorithm thus either ends up in the unique **terminate** state that corresponds to successful completion, or else in the **abort** state. □

All states of the Markov chain may be projected onto their buffer $\{\text{next}_a, \text{next}_b\}$ and visualized (see Fig. 4)).

Remark 9 (CAS statement)**.** *The CAS statements (see* 10ι *in Alg. 5) acquire their full meaning in the multithreaded Alg. 6 The way in which they differ from simple* **if** *statements can already be illustrated in the simplified setting. We suppose two threads* a *and* b*. Then, with* j_ι *the target sphere on thread* ι*, a buffer content* $\{10a, 10b\}$*:*

⋮	⋮		⋮	⋮
$10a$	$j_a.tag.CAS(static, a)$		$10b$	$j_b.tag.CAS(static, b)$
$11a$	**if** $j_a.tag = a$ **:**		$11b$	**if** $j_b.tag = b$ **:**
⋮	⋮		⋮	⋮

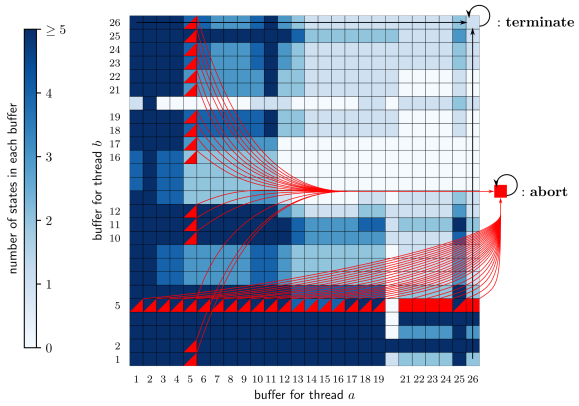

Figure 4: The 3670 states in Alg. 5 for the `SequentialC4` test suite projected onto the buffer content $\{\text{next}_a, \text{next}_b\}$ (see Fig. 3). The **terminate** buffer $\{26a, 26b\}$ corresponds to a single state.

can belong to a state with $j_a = 3 = j_b = 3$.[3] *If the statement* $10a$ *is selected, sphere* 3 *becomes active on thread* a *(through the statement* $3.tag = a$*). In contrast, if the switch selects* $10b$*, sphere* 3 *becomes active on thread* b*. The program continues consistently for both switch choices, because the selection is made in a single ("atomic") step on each thread and because the sequential-consistency model avoids conflicting memory assignments. In contrast, if the switch selection from* $\{10a, 10b\}$ *is split as:*

$\vdots$	$\vdots$	$\vdots$	$\vdots$
$10'a$	**if** $j_a.tag = static$ **:**	$10'b$	**if** $j_b.tag = static$ **:**
$10''a$	$j_a.tag = a$	$10''b$	$j_b.tag = b$
$11a$	**if** $j_a.tag = a$ **:**	$11b$	**if** $j_b.tag = b$ **:**
$\vdots$	$\vdots$	$\vdots$	$\vdots$

the sequence $10'a \to 10'b \to 10''a \to 11a \to 10''b \to 11b$ *results in sphere* 3 *first becoming active on thread* a *(and the thread continuing as if this remained the case), and then on thread* b*, which is inconsistent. In Algorithm 6, the C++ implementation of multithreaded ECMC, the CAS likewise keeps this selection step atomic, and likewise excludes memory conflicts among all threads during this step. It thus plays the role of a lightweight memory lock.*

Algorithm 5 features lock-free programming, which is also a key ingredient of Algorithm 6.

[3] This corresponds to the lifted configuration of Fig. 3h

Remark 10 (Lock-free programming)**.** *To illustrate lock-free programming in Alg. 5, we consider two threads, a and b.*

$7a$	$j_a \leftarrow \tilde{j}$	$\vdots$	$\vdots$
$8a$	$x_a \leftarrow x_{\tilde{j}}$	$16b$	$i_b.x \leftarrow i_b.x + \tau_b$
$\vdots$	$\vdots$	$17b$	$i_b.tag \leftarrow static$
$10a$	$j_a.tag.CAS(static, a)$	$\vdots$	$\vdots$
$\vdots$	$\vdots$		
$13a$	**if** $x_a = j_a.x$ **:**		
$\vdots$	$\vdots$		

The identification of the target sphere j_a on thread a (statements $7a$ and $8a$) would be compromised if, before locking through the CAS statement at $10a$, it was changed in thread b, where the same sphere i_b is active (see statements $16b, 17b$). However, the statement $13a$ checks that sphere j_a has not moved. If this condition is not satisfied, the thread a will be restarted (through statement $20a$). (See also Remark 12.)

2.4. Multithreaded ECMC (C++, OpenMP implementation)

Algorithm 6, discussed in this section, translates Algorithm 5 into C++ (OpenMP). The CAS statement and lock-free programming assure its efficiency. A sphere's attributes are again its position, its local time, and its tag. The latter is an atomic variable. We refer to line numbers in Algorithm 5.

Algorithm 6 (Multithreaded ECMC (C++, OpenMP))**.** *With initial values as in Alg. 1, thread management is handled by OpenMP. The number of threads can be smaller than the number of active spheres. The multithreading stage transliterates the one of Alg. 5. Statement 2ι of Alg. 5 is implemented through a constraint graph (see Section 3.1). Statements 10ι through 13ι are expressed as follows in* `MultiThreadECMC.cc`*:*

```
10ι →  j->tag.compare_exchange_strong(...static,...
11ι →  if (j->tag.load(memory_order) == iota)
12ι →  if (tau < distance)
13ι →  if (x == j->x),
```

where the `memory_order` *qualifier may take on different values (see Section 3.2). Important differences with Alg. 5 are discussed in Remarks 11 and 12. Output is as for Alg. 1.*

Code availability. *Alg. 6 is implemented in* `MultiThreadECMC.cc`*. It is executed in several validation and benchmarking scripts (see Section 3.2).*

Remark 11 (Active-sphere necklaces)**.** *Alg. 5 restarts thread ι if the target sphere j (for an active sphere i on the thread) is itself active on another thread. With periodic boundary conditions, active-sphere necklaces, where all target spheres are active, can deadlock the algorithm. To avoid this, Alg. 6 moves sphere i up to contact with j before restarting (this is also used in Alg. 4).*

The source code of Algorithm 6 essentially translates that of Algorithm 5. The compiler may however change the order of execution for some statements in order to gain efficiency. (The memory access in modern multi-core processors can also be very complex and, in particular, thread-dependent.) Attributes, such as the `memory_order` qualifier in the CAS statement, may constrain the allowed changes of order. The reordering directives adopted in Algorithm 6 were chosen and validated with the help of extensive runs from randomly generated configurations. However, subtle pitfalls escaping notice through such testing can be exposed by explicitly reordering statements in Algorithm 5.

Remark 12 (Memory-order directives in Algs 5 and 6)**.** *In the* `SequentialC5` *test suite with* $N = 5$*, interchanging statements* 15ι *and* 16ι *in Alg. 5 yields a spurious absorbing state, and invalidates the algorithm. The same test suite can also be input into Alg. 6, where it passes the* `Ordering.sh` *validation test, even if the statements in* `MultiCPP.cc` *corresponding to* 15ι *and* 16ι *are exchanged. However, a* $1\mu s$ `pause` *statement introduced in the C++ program between what corresponds to the (interchanged) statements* 15ι *and* 16ι *produces a* $\sim 1\%$ *error rate, illustrating that Alg. 6 is unsafe without a protection of the order of the said statements. Safety may be increased through atomic position and local time variables, allowing the use of the* `fetch_add()` *operation to displace spheres.*

3. Tools, validation protocols, benchmarks, and extensions

We now discuss the implementations of the algorithms of Section 2, as well as their validation protocols, benchmarks, and possible extensions. We also discuss the prospects of this method beyond this paper's focus on the interval between two breakpoints h and h'.

In our implementation of Algorithms 2, 4, and 6, a directed constraint graph encodes the possible pairs of active and target spheres as arrows $[i \to j]$ (see Section 3.1). The outdegree of this graph is at most three, and a rough constraint graph $\mathcal{G}^{(3)}$ with, usually, outdegree three for all vertices is easily generated. $\mathcal{G}^{(3)}$ may contain redundant arrows that cannot correspond to liftings. Our pruning algorithm eliminates many of them. We also prove that $\mathcal{G}^{\min}$, the minimal constraint graph, is planar. This may be of importance if disjoint parts of the constraint graph are stored on different CPUs, each with a number of dedicated threads. In general, we expect constraint graphs to be a useful tool for hard-sphere production codes, with typically $\mathcal{O}(N)$ liftings between changes of $\mathbf{v}$.

Validation scripts are discussed in Section 3.2. Scripts check that the liftings of standard cell-based ECMC are all accounted for in the used constraint graph. For the ECMC algorithms of Section 2, the set $\mathcal{L}$ of liftings provides the complete history of each run, and scripts check that they correspond to $\mathcal{L}^{\mathrm{ref}}$.

In Section 3.3, we benchmark Algorithm 6 and demonstrate a speed-up by an order of magnitude for a single CPU with 40 threads on an x86 CPU (see Section 3.3). The overhead introduced by multithreading (~ 2.4) is very reasonable. We then discuss possible extension of our methods (see Section 3.4).

3.1. Constraint graphs

For a given initial condition $\mathcal{C}_h$ and velocity $\mathbf{v}$, arrows $[i{\to}j]$ of the constraint graph $\mathcal{G}_{\mathbf{v}}$ represent possible liftings $l_t = ([i{\to}j], (\mathbf{x}, \mathbf{x}'), t)$ [26]. Arrows remain unchanged between breakpoints because spheres i and j with a perpendicular distance of less than 2σ cannot hop over one another (this argument can be adapted to periodic boundary conditions), and pairs with larger perpendicular distance are absent from $\mathcal{G}$. All constraint graphs $\mathcal{G}_{\mathbf{v}}$ are supersets of a minimal constraint graph $\mathcal{G}^{\min}_{\mathbf{v}} \equiv \mathcal{G}^{\min}_{-\mathbf{v}}$ (where the equivalence is understood as $[i{\to}j]_{\mathbf{v}} \equiv [j{\to}i]_{-\mathbf{v}}$).

Remark 13 (Constraint graphs and convex polytopes)**.** *Each arrow $[i{\to}j]$ of the constraint graph $\mathcal{G}_{\mathbf{v}}$ provides (for $\mathbf{v} = (1, 0)$) an inequality*

$$x_i \leq x_j - b_{ij} \tag{3}$$

that is tight ($x_i = x_j - b_{ij}$) when i lifts to j at contact (if there are configurations $\mathcal{C}$ where it is tight, then $[i{\to}j]$ belongs to $\mathcal{G}^{\min}_{\mathbf{v}}$). The set of inequalities defines a convex polytope. With periodic boundary conditions (unaccounted for in eq. (3)*), this polytope is infinite in the direction corresponding to uniform translation of all spheres with $\mathbf{v}$ (see [26]).*

Remark 14 (Constraint graphs and irreducibility)**.** *Rigorously, we define the constraint graph $\mathcal{G}^{\min}_{\mathbf{v}}$ as the set of arrows $[i{\to}j]$ that are encountered from $\mathcal{C}_h$ by Alg. 1 (or, equivalently, Alg. 2) at liftings l_t $\forall t \in (-\infty, \infty)$. The liftings for $t < 0$ can be constructed because of time-reversal invariance (see Remark 3). For the same reason, we have $\mathcal{G}^{\min}_{\mathbf{v}} \equiv \mathcal{G}^{\min}_{-\mathbf{v}}$, and the set of arrows reached from $\mathcal{C}_h$ is equivalent to that reached from any configuration that is reached from $\mathcal{C}$ (and in particular $\mathcal{C}_{h'}$). While we expect ECMC to be irreducible in the polytope defined through the inequalities in eq.* (3)*, we do not require irreducibility for the definition of $\mathcal{G}^{\min}$.*

Between breakpoints, the active sphere i can lift to at most three other spheres, namely the sphere j^0 minimizing the time of flight τ_{ij} in a corridor of width 2σ around the center of i , and likewise the closest-by sphere j^+ in the corridors $[\sigma, 2\sigma]$ and sphere j^- in the corridor $[-2\sigma, -\sigma]$ (see Fig. 5a). The set of arrows $\{[i{\to}j^0], [i{\to}j^-], [i{\to}j^+]\ \forall i \in \{1, \ldots, N\}\}$ constitutes the constraint graph $\mathcal{G}^{(3)}_{\mathbf{v}}$, which is thus easily computed. Except for small systems (where the corridors may be empty), $\mathcal{G}^{(3)}$ has outdegree three for all spheres i. However, its indegree is not fixed. The constraint graph $\mathcal{G}^{(3)}$ is not necessarily locally planar,[4] and in the embedding provided by the sphere centers of a given configuration, non-local arrows can be present (see Fig. 6a). However, $\mathcal{G}^{\min}$ can be proven to be locally planar (see Fig. 6b)).

Lemma 4. *The graph $\mathcal{G}^{\min}$ is locally planar, and any sphere configuration that can be reached between breakpoints provides a locally planar embedding.*

[4] "Locally planar" means that any subgraph that does not sense the periodic boundary conditions is planar.

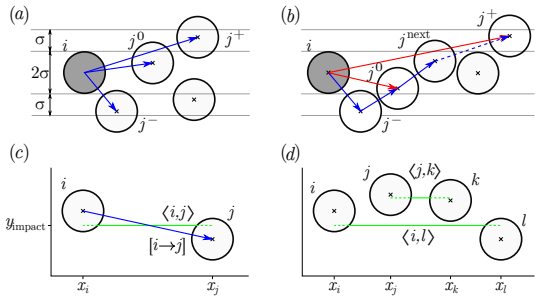

Figure 5: Constraint graphs, pruning, and planarity. (a): Corridors of an active sphere i, with arrows $[i{\to}j^-]$, $[i{\to}j^0]$, and $[i{\to}j^+]$ belonging to $\mathcal{G}^{(3)}_{(1,0)}$. (b): Pruning of an arrow $[i{\to}j^+]$ through a sphere j^{next} without there being an arrow $[i{\to}j^{\text{next}}]$. (c): Spheres i and j, arrow $[i{\to}j]$, and impact path $\langle i,j\rangle$. Other spheres cannot cover $\langle i,j\rangle$. (d): Spheres i,j,k,l with $x_i < \cdots < x_l$ and impact paths $\langle i,l\rangle$ and $\langle j,k\rangle$.

Proof. We first consider two spheres i and j for $\mathbf{v} = (1,0)$ in the plane (without taking into account periodic boundary conditions). The arrow $[i{\to}j]$ is drawn by connecting the centers of i and j. The impact path $\langle i,j\rangle$ is the horizontal line segment connecting (x_i, y^{impact}) and (x_j, y^{impact}) where y^{impact} is the vertical position at which the two spheres can touch by moving them with $\mathbf{v}$ (see Fig. 5c). If the arrow $[i{\to}j]$ exists, no other sphere can intersect the impact path $\langle i,j\rangle$.

For four spheres i,j,k,l, we now show that no two arrows between spheres can cross each other. The x-values can be ordered as $x_i < x_j < x_k < x_l$ (again without taking into account periodic boundary conditions). Two arrows between three spheres trivially cannot cross. For arrows between two pairs of spheres, arrows $[i{\to}j]$ and $[k{\to}l]$ cannot cross. Likewise, if there is an arrow $[i{\to}k]$, then sphere j must be on one side of the impact path $\langle i,k\rangle$, and k must be on the other side of $\langle j,l\rangle$, so that arrows $[i{\to}k]$ and $[j{\to}l]$ cannot cross. Finally, if arrow $[i{\to}l]$ exists, then j and k must be on the same side of the impact path $\langle i,l\rangle$ in order to have an impact path. But then, $[j{\to}k]$ cannot cross $[i{\to}l]$ (see Fig. 5d). □

The minimal constraint graph $\mathcal{G}^{\min}$ is more difficult to compute than $\mathcal{G}^{(3)}$ because the underlying "redundancy detection" problem is not strictly polynomial in system size, although practical algorithms exist [27]. However, $\mathcal{G}^{(3)}$ can be pruned of redundant constraints that correspond to pairs of spheres i and j that are prevented from lifting by other spheres. For example, given arrows $[i{\to}j]$, $[j{\to}k]$ and $[i{\to}k]$, the latter can be "first-order" pruned (eliminated with one intermediary, namely j) if $b_{ij} + b_{jk} > b_{ik}$ (in Fig. 5, $[i{\to}j^+]$ can be pruned for this reason). The presence of the arrow $[j{\to}k]$ is not necessary to make this argument work (see Fig. 5b). Pruning can be taken to higher orders. To second order, if $b_{ij} + b_{jk} + b_{kl} > b_{il}$, then the arrow $[i{\to}l]$ can be eliminated. Finally, any arrow $[i{\to}j]$ in $\mathcal{G}_{\mathbf{v}}$ can be pruned through symmetrization if it is unmatched by $[j{\to}i]$ in $\mathcal{G}_{-\mathbf{v}}$ because $\mathcal{G}^{\min}_{\mathbf{v}} \equiv \mathcal{G}^{\min}_{-\mathbf{v}}$, with $\mathcal{G}[\mathbf{v}]$ and $\mathcal{G}[-\mathbf{v}]$ obtained separately

(see Remark 3).

Rarely, arrows can be eliminated by symmetrizing graphs that were pruned to third or fourth order, and constraint graphs that are obtained in this way appear close to $\mathcal{G}^{\min}$ (see Section 3.2).

Code availability. *The constraint graph* $\mathcal{G}^{(3)}$ *is constructed in* `GenerateG3.py` *and pruned to* $\mathcal{G}$ *in* `PruneG.py`. *The program* `GraphValidateCellECMC.cc` *runs cell-based ECMC to verify the consistency of* $\mathcal{G}$.

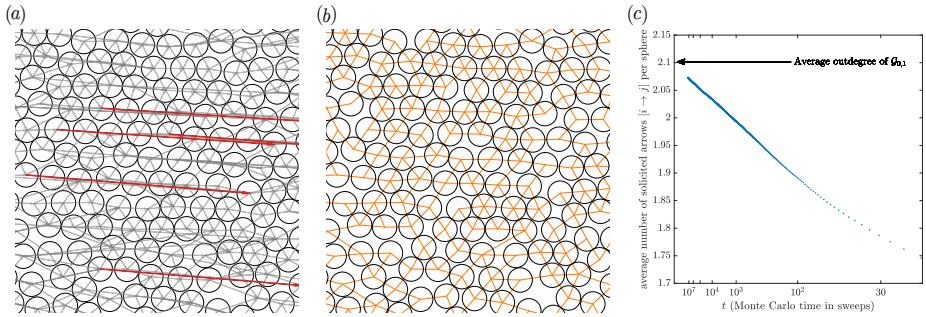

Figure 6: ECMC constraint graphs for $\mathcal{C}^{256}$ (see Section 3.2 for definition). (a): $\mathcal{G}^{(3)}$ for this configuration (detail), with highlighted non-local arrows. (b): $\mathcal{G}^{256}$ (same detail), obtained from $\mathcal{G}^{(3)}$ through fourth-order pruning followed by symmetrization. (c): Number of solicited arrows in $\mathcal{G}^{256}$ in a long cell-based ECMC run, compared to its average outdegree.

3.2. Validation

Our programs apply to arbitrary density $\eta = N\pi\sigma^2/L^2$ and linear size L of the periodic square box (with $N = M^2$). We provide sets of configurations and constraint graphs for validation and benchmarking. One such set consists in a configuration $\mathcal{C}^{256}$ at $M = 256$, and for $\eta = 0.708$ and a fourth-order symmetrized constraint graph $\mathcal{G}^{256}$. Where applicable, the number of active spheres varies as $k = 1, 2, 4, \ldots, k^{\max}$ and the number of threads as $n_\iota = 1, 2, 3, \ldots, n_\iota^{\max}$. For fixed k and n_ι, there are n_{run} runs that vary h and h'.

Constraint-graph validation. Constraint graphs are generated in the `Setup.sh` script. The `GraphValidateCellECMC` test performs cell-based ECMC derived from `CellECMC.f90` [3, 23], where spheres are assigned to local cells and neighborhood-cell searches identify possible liftings. Cell-based ECMC must exclusively solicit liftings accounted for in $\mathcal{G}$. The `GraphValidateCellECMC` test also records the sweep (lifting per sphere) at which an arrow $[i \to j] \in \mathcal{G}$ is first solicited in a lifting and compares the time evolution of the average number of solicited arrows with its average outdegree. The $\mathcal{G}^{256}$ constraint graph passes the validation test with $t = 2\times10^7$ sweeps. The outdegree of $\mathcal{G}^{256}$ is 2.1, and 98.7 % of its arrows are solicited during the test. Logarithmic extrapolation (with $1/(\ln t)^\alpha$, $\alpha = 1.7$) suggests that $\mathcal{G}^{256}$ essentially agrees with $\mathcal{G}^{\min}$ (see Fig. 6c).

Use of $\mathcal{G}^{256}$ rather than $\mathcal{G}^{(3)}$ speeds up ECMC, but further performance gains through additional pruning are certainly extremely limited.

Validation of Algs 4 and 6. Our implementations of Algorithms 4 and 6 are modified as discussed in Remark 7. Runs compute the set $\mathcal{L}_{t^*}$ to the earliest horizon-violation time t^* (with $t^* = h'$ if the run concludes successfully). The `PValidateECMC.sh` test first advances $\mathcal{C}^{256} = \mathcal{C}_{t=0}$ to a random breakpoint h (using $\mathcal{L}^{\mathrm{ref}}$). Each test run is in the interval $[h, h']$, where h' is randomly chosen. To pass the validation test, $\mathcal{L}_{t^*}$ must for each run agree with $\mathcal{L}^{\mathrm{ref}}$ (see Section 4 for details of scripts used). Algorithm 4 passes the `PValidateECMC.sh` test with $n_{\mathrm{run}} = 1\times10^3$ for $k^{\max} = 8192$.

Our x86 computer has two Xeon Gold 6230 CPUs with variable frequency from 2.1 GHz to 3.9 GHz, each with 20 cores and 40 hardware threads. We use OpenMP directives to restrict all threads to a single CPU. We consider again $\mathcal{C}^{256} = \mathcal{C}_{t=0}$ as the initial configuration, and then run the program from h to h'. On our x86 CPU, Algorithm 6 passes the `CValidateECMC.sh` test with $n_{\mathrm{run}} = 1\times10^3$, $k^{\max} = 8192$ and $n_\iota^{\max} = 40$.

On our ARM CPU (Nvidia Jetson with Cortex A57 CPU (at 1.43 GHz) with four cores and four hardware threads), we again consider $\mathcal{C}^{256} = \mathcal{C}_{t=0}$ as initial configuration. For the same system parameters as above, Algorithm 6 passes the `CValidateECMC.sh` test with $n_{\mathrm{run}} = 1\times10^3$, $k^{\max} = 8192$ and $n_\iota^{\max} = 4$. The ARM architecture allows dynamic re-ordering of operations, and the separate validation test more severely scrutinizes thread interactions than for the x86 CPU.

On both CPUs, Algorithm 6 passes the `CValidateECMC.sh` test with the following choices of `memory_order` directives:

`memory_order_relaxed`. This most permissive memory ordering of the C++ memory model imposes no constraints on compiler optimization or dynamic re-ordering of operations by the processors, and only guarantees the atomic nature of the CAS operation. Such re-orderings are more aggressive on ARM CPUs than on x86 CPUs. This memory ordering does not guarantee that the statements constituting the lock-less lock are executed as required (see Remark 12).

`memory_order_seq_cst` for all memory operations on the tag attribute. This directive imposes the sequential-consistency model (see Remark 12) for each access of the tag attribute. It slows down the code by 40% compared to the `memory_order_relaxed` directive.

`memory_order_acquire` on `load`, `memory_order_release` on `store`. This directive implies "acquire–release" semantics on the tag attribute. It imposes a lock-free exchange at each operation on the tag attribute, so that all variables, including positions and local times, are synchronized between threads during tag access. CAS remains `memory_order_seq_cst`. This directive maintains speed compared to `memory_order_relaxed`, yet provides better guarantees on the propagation of variable modification

between threads. `MultiThreadECMC.cc` compiles by default with this directive.

3.3. Benchmarks for Algorithm 6 (x86 and ARM)

Algorithm 6 is modified as discussed in Remark 7 (program execution continues in spite of horizon violations) and used for large values of h'. This measures the net cost of steady-state thread interaction, without taking into account thread-setup times. We report here on results of the `BenchmarkECMC.sh` script for $\mathcal{C}^{256}$ as an initial configuration and $\mathcal{G}^{256}$ with $k = 40$ active spheres. The number of threads varies as $n_\iota = 1, 2, 3, \ldots, n_\iota^{\max}$, with $n_{\text{run}} = 20$.

On our x86 CPU (see Section 3.2), the `BenchmarkECMC.sh` script is parametrized with $n_\iota^{\max} = 40$. The benchmark speed increases roughly linearly up to 20 threads (reaching a speed-up of 10 for 20 threads), and then keeps improving more slowly with a maximum for 40 threads at a speed-up of 14 and an absolute speed of $\sim 1.6 \times 10^{12}$ events/hour (see Fig. 7). The variable frequency of Xeon processors under high load may contribute to this complex behavior. On a single thread, our program runs 2.2 times slower than an unthreaded code, due to the eliminated overhead from threading constructs. The original `CellECMC.f90` cell-based production code generates 3×10^{10} events/hour. The use of a constraint graph, rather than a cell-based search, thus improves performance by almost an order of magnitude, if the set-up of $\mathcal{G}$ is not accounted for.

On our ARM CPU, the `BenchmarkECMC.sh` script is parametrized with $n_\iota^{\max} = 4$. The benchmark speed increases as the number of threads, reaching a speed-up of 3.8 for $n_\iota = 4$. The absolute speed is about six times smaller than for our x86 CPU for a comparable number of threads, as may be expected for a low-power processor designed for use in mobile phones.

3.4. Birthday problem, full ECMC, multi-CPU extensions

In this section, we treat some practical aspects for the use of Algorithm 6.

Birthday problem. We analyze multithreaded ECMC in terms of the (generalized) birthday problem, which considers the probability p that two among k' integers (modeling individuals) sampled from a discrete uniform distribution in the set $\{1, 2, \ldots, N'\}$ (modeling birthdays) are the same. For large N', $p \sim [1 - \exp(-k'^2/(2N'))]$ [28], which is small if $k' \lesssim \sqrt{N'}$. At constant density η, sphere radius σ, velocity $\mathbf{v}$, and time interval $h' - h$, each active chain ι is almost restricted to a region of constant area, whereas the total area of the simulation box is ηN. We may suppose that the k active spheres are randomly positioned in the simulation box broken up into a grid of $\propto N$ constant-area cells. For $k \lesssim \sqrt{N}$, we expect the probability that one of these cells contains two active spheres to remain constant for $N \to \infty$, and therefore also the probability of an update-order violation for constant $h' - h$.

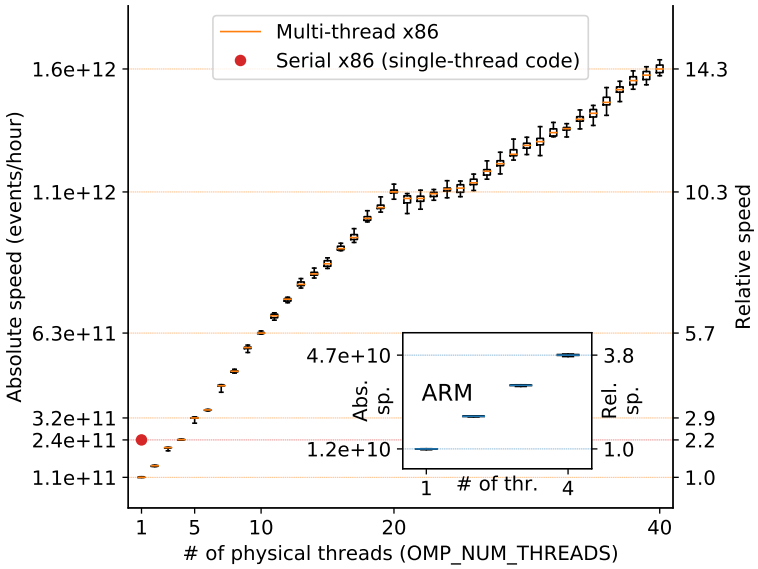

Figure 7: Output of the `BenchmarkECMC.sh` script for Algorithm 6 (five-number summary of 20 runs) for $k = 80$ on an x86 CPU with 20 cores and $1, \ldots, 40$ threads, and for serial code that processes active chains sequentially. Inset: Output of the script for a four-core ARM CPU.

Restarts. Our algorithms reproduce output of Algorithm 2 only if they do not abort. In production code, the effects of horizon violations will have to be repaired. Two strategies appear feasible. First, the algorithm may restart the run from a copy of $\{\mathcal{C}_h, \mathcal{A}_h\}$ at the initial breakpoint h, and choose a smaller breakpoint h'', for example the time of abort. The successful termination of this restart is not guaranteed, as the individual threads may organize differently. Second, the time evolution may be reconstructed from $\mathcal{L}_{t^*}$ to the earliest horizon-violation time t^* (see Remark 7), and t^* may then be used as the subsequent initial breakpoint. Besides an efficient restart strategy, a multithreaded production code will also need an efficient parallel algorithm for computing $\mathcal{G}$ after a change of $\mathbf{v}$.

Multi-CPU implementations. Algorithm 6 is spelled out for a single shared-memory CPU and for threads that may access attributes of all spheres (see statement 10ι in Algorithm 5 and Remark 11). However, thread interactions are local and immutable in between breakpoints (as evidenced by the constraint graphs). This invites generalizations of the algorithm to multiple CPUs (each of them with many threads). Most simply, two CPUs could administer disjoint parts of the constraint graph, for example with interface vertices doubled up on both of them (see Fig. 8). In this way, an active sphere arriving at an interface would simply be copied out to the neighboring CPU. The generalization to multiple CPUs appears straightforward.

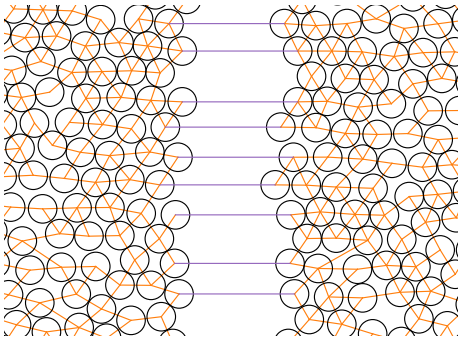

Figure 8: A constraint graph doubled up for a multi-CPU implementation of Alg. 6 (detail of $\mathcal{C}^{256}$ configuration shown). Interface vertices appear on both sides.

4. Available computer code

All implemented algorithms and used scripts that are made available on GitHub in `ParaSpheres`, a public repository which is part of a public GitHub organization.[5] Code is made available under the GNU GPLv3 license (for details see the `LICENSE` file).

The repository can be forked (that is, copied to an outside user's own public repository) and from there studied, modified and run in the user's local environment. Users may contribute to the `ParaSpheres` project *via* pull requests (see the `README.md` and `CONTRIBUTING.md` files for instructions and guidelines). All communication (bug reports, suggestions) take place through GitHub "Issues", that can be opened in the repository by any user or contributor, and that are classified in GitHub projects.

Implemented algorithms. The following programs are located in the directory tree under their language (`F90`, `Python` or `CPP`) and in similarly named subdirectories, that all contain `README` files for further details. Some of the longer programs are split into modules.

Code/Directory	Algorithm / Usage
`CellECMC.f90`	Cell-based production ECMC [3]
`GenerateG3.py`	Generate $\mathcal{G}^{(3)}$ (Section 3.1)
`PruneG.py`	Prune $\mathcal{G}$ (Section 3.1)
`GraphValidateCellECMC.cc`	Validate $\mathcal{G}$ against cell-based ECMC
`GlobalTimeECMC.py`	Alg. 2 (Section 2.1)
`SingleThreadLocalTimeECMC.py`	Alg. 4 (Section 2.2)
`SequentialMultiThreadECMC.py`	Alg. 5 (Section 2.3)
`MultiThreadECMC.cc`	Alg. 6 (Section 2.4)

[5] The organization's url is `https://github.com/jellyfysh`.

Scripts and validation suites. The `Scripts` directory provides the following bash scripts to compile and run groups of programs and to reproduce all our results:

Script	Summary of usage
`Setup.sh`	Prepare $\mathcal{C}_{t=0}$, $\mathcal{G}$, $\mathcal{L}^{\mathrm{ref}}$
`SequentialC4.sh`	Test suite for Alg. 5 with $N = 4$
`SequentialC5.sh`	Test suite for Alg. 5 with $N = 5$ (see Remark 12)
`Ordering.sh`	Test suite for Alg. 6 with $N = 5$ (see Remark 12)
`ValidateG.sh`	Validate constraint graph
`PValidateECMC.sh`	Validate Alg. 4 against $\mathcal{L}^{\mathrm{ref}}$
`CValidateECMC.sh`	Validate Alg. 6 against $\mathcal{L}^{\mathrm{ref}}$
`BenchmarkECMC.sh`	Benchmark `MultiThreadECMC.cc`, generate Fig. 7

In the `Setup.sh` script, `CellECMC.f90` first produces an sample $\mathcal{C}_0$ such that the unidirectional dynamics in $\mathcal{C}_0$ is practically aperiodic. It then generates $\mathcal{G}^{(3)}$ with `GenerateG3.py`, and runs `PruneG.py` to output $\mathcal{G}$. Finally, it runs `GlobalTimeECMC.py` for each set $\mathcal{A}_0$, in order to generate generate several $\mathcal{L}^{\mathrm{ref}}$. The `ValidateG.sh` script uses `GraphValidateCellECMC.cc` to run cell-based ECMC, and verifies that all liftings are accounted for in $\mathcal{G}$. It also tracks the solicitation of arrows as a function of time. The `PValidateECMC.sh` script validates `SingleThreadLocalTimeECMC.py` by comparing the sets of liftings with $\mathcal{L}^{\mathrm{ref}}$ from `Setup.sh`. The `CValidateECMC.sh` script does the same for `MultiThreadECMC.cc`. `BenchmarkECMC.sh` benchmarks `MultiThreadECMC.cc` for different numbers of threads. The test suites are concerned with small-N configurations.

5. Conclusions and outlook

In this paper, we presented an event-driven multithreaded ECMC algorithm for hard spheres which enforces thread synchronization at infrequent breakpoints only. Between breakpoints, spheres carry and update local times. Possible inconsistencies are locally detected through a horizon condition. Within ECMC, our method avoids the scheduling problem that has historically plagued event-driven molecular dynamics. This is possible because in ECMC only few spheres move at any moment, and all have the same velocity. Conflicts are thus exceptional, and little information is exchanged between threads. We relied on the generalized birthday problem to show that our algorithm remains viable up to a number of threads that grows as the square root of the number of spheres, a setting relevant for the simulation of millions of spheres for modern commodity servers with ~ 100 threads. The mapping of Algorithm 5 onto an absorbing Markov chain allowed us to prove its correctness (for a given lifted initial configuration) and to rigorously analyze side effects of code re-orderings in the multithreaded C++ code.

Our algorithm is presently implemented between two global breakpoint times, where it achieves considerable speed-up with respect to sequential ECMC. A

fully practical multithreaded ECMC code that greatly outperforms cell-based algorithms appears within reach. It is still a challenge to understand whether multithreaded ECMC applies to general interacting-particle systems.

Acknowledgements

W.K. acknowledges support from the Alexander von Humboldt Foundation. We thank E. P. Bernard for allowing his original hard-sphere ECMC production code to be made available.

References

[1] E. P. Bernard, W. Krauth, D. B. Wilson, Event-chain Monte Carlo algorithms for hard-sphere systems, Phys. Rev. E 80 (2009) 056704. doi:10.1103/PhysRevE.80.056704.
URL http://link.aps.org/doi/10.1103/PhysRevE.80.056704

[2] M. Michel, S. C. Kapfer, W. Krauth, Generalized event-chain Monte Carlo: Constructing rejection-free global-balance algorithms from infinitesimal steps, J. Chem. Phys. 140 (5) (2014) 054116. arXiv:1309.7748, doi:10.1063/1.4863991.

[3] E. P. Bernard, W. Krauth, Two-Step Melting in Two Dimensions: First-Order Liquid-Hexatic Transition, Phys. Rev. Lett. 107 (2011) 155704. doi:10.1103/PhysRevLett.107.155704.
URL http://link.aps.org/doi/10.1103/PhysRevLett.107.155704

[4] S. C. Kapfer, W. Krauth, Two-Dimensional Melting: From Liquid-Hexatic Coexistence to Continuous Transitions, Phys. Rev. Lett. 114 (2015) 035702. doi:10.1103/PhysRevLett.114.035702.
URL http://link.aps.org/doi/10.1103/PhysRevLett.114.035702

[5] M. Hasenbusch, S. Schaefer, Testing the event-chain algorithm in asymptotically free models, Phys. Rev. D 98 (2018) 054502. doi:10.1103/PhysRevD.98.054502.
URL https://link.aps.org/doi/10.1103/PhysRevD.98.054502

[6] E. P. Bernard, W. Krauth, Addendum to "Event-chain Monte Carlo algorithms for hard-sphere systems", Physical Review E 86 (1). doi:10.1103/physreve.86.017701.
URL https://doi.org/10.1103/physreve.86.017701

[7] S. C. Kapfer, W. Krauth, Irreversible Local Markov Chains with Rapid Convergence towards Equilibrium, Phys. Rev. Lett. 119 (2017) 240603. doi:10.1103/PhysRevLett.119.240603.
URL https://link.aps.org/doi/10.1103/PhysRevLett.119.240603

[8] M. F. Faulkner, L. Qin, A. C. Maggs, W. Krauth, All-atom computations with irreversible Markov chains, The Journal of Chemical Physics 149 (6) (2018) 064113. doi:10.1063/1.5036638.
URL https://doi.org/10.1063/1.5036638

[9] J. Harland, M. Michel, T. A. Kampmann, J. Kierfeld, Event-chain Monte Carlo algorithms for three- and many-particle interactions, EPL (Europhysics Letters) 117 (3) (2017) 30001.
URL http://stacks.iop.org/0295-5075/117/i=3/a=30001

[10] 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. doi:10.1063/1.1699114.

[11] B. J. Alder, T. E. Wainwright, Phase Transition for a Hard Sphere System, J. Chem. Phys. 27 (1957) 1208–1209. doi:10.1063/1.1743957.

[12] B. J. Alder, T. E. Wainwright, Studies in Molecular Dynamics. I. General Method, J. Chem. Phys. 31 (1959) 459–466. doi:10.1063/1.1730376.

[13] D. C. Rapaport, The Event Scheduling Problem in Molecular Dynamic Simulation, Journal of Computational Physics 34 (1980) 184–201. doi:10.1016/0021-9991(80)90104-7.

[14] M. Isobe, Hard sphere simulation in statistical physics methodologies and applications, Molecular Simulation 42 (16) (2016) 1317–1329. doi:10.1080/08927022.2016.1139106.

[15] B. D. Lubachevsky, Simulating billiards serially and in parallel, International Journal in Computer Simulation 2 (1992) 373–411.

[16] B. Lubachevsky, Several unsolved problems in large-scale discrete event simulations, ACM SIGSIM Simulation Digest 23 (1993) 60–67. doi:10.1145/174134.158467.

[17] A. G. Greenberg, B. D. Lubachevsky, I. Mitrani, Superfast parallel discrete event simulations, ACM Transactions on Modeling and Computer Simulation (TOMACS) 6 (2) (1996) 107–136.

[18] A. T. Krantz, Analysis of an efficient algorithm for the hard-sphere problem, ACM Trans. Model. Comput. Simul. 6 (3) (1996) 185–209. doi:10.1145/235025.235030.

[19] M. Marin, Billiards and related systems on the bulk-synchronous parallel model, in: Proceedings 11th Workshop on Parallel and Distributed Simulation, 1997, pp. 164–171. doi:10.1109/PADS.1997.594602.

[20] S. Miller, S. Luding, Event-driven molecular dynamics in parallel, Journal of Computational Physics 193 (1) (2004) 306 – 316. arXiv:physics/0302002, doi:10.1016/j.jcp.2003.08.009.

[21] P. Diaconis, S. Holmes, R. M. Neal, Analysis of a nonreversible Markov chain sampler, Annals of Applied Probability 10 (2000) 726–752.

[22] D. A. Levin, Y. Peres, E. L. Wilmer, Markov Chains and Mixing Times, American Mathematical Society, 2008.

[23] M. Engel, J. A. Anderson, S. C. Glotzer, M. Isobe, E. P. Bernard, W. Krauth, Hard-disk equation of state: First-order liquid-hexatic transition in two dimensions with three simulation methods, Phys. Rev. E 87 (2013) 042134. doi:10.1103/PhysRevE.87.042134.
URL http://link.aps.org/doi/10.1103/PhysRevE.87.042134

[24] Lamport, How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, IEEE Transactions on Computers C-28 (9) (1979) 690–691. doi:10.1109/tc.1979.1675439.

[25] H. J. Boehm, L. Crowl, C++ atomic types and operations, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html (2009).

[26] S. C. Kapfer, W. Krauth, Sampling from a polytope and hard-disk Monte Carlo, Journal of Physics: Conference Series 454 (1) (2013) 012031. doi:10.1088/1742-6596/454/1/012031.
URL http://stacks.iop.org/1742-6596/454/i=1/a=012031

[27] K. Fukuda, B. Gärtner, M. Szedlák, Combinatorial redundancy detection, Annals of Operations Research 265 (1) (2016) 47–65. doi:10.1007/s10479-016-2385-z.

[28] F. H. Mathis, A generalized birthday problem, SIAM Review 33 (2) (1991) 265–270. doi:10.1137/1033051.