Efficient event-driven simulations of hard spheres

Frank Smallenburg

Efficient event-driven simulations of hard spheres

Frank Smallenburg
Université Paris-Saclay, CNRS, Laboratoire de Physique des Solides, 91405 Orsay, France

Hard spheres are arguably one of the most fundamental model systems in soft matter physics, and hence a common topic of simulation studies. Event-driven simulation methods provide an efficient method for studying the phase behavior and dynamics of hard spheres under a wide range of different conditions. Here, we examine the impact of several optimization strategies for speeding up event-driven molecular dynamics of hard spheres, and present a light-weight simulation code that outperforms existing simulation codes over a large range of system sizes and packing fractions. The presented differences in simulation speed, typically a factor of five to ten, save significantly on both CPU time and energy consumption, and may be a crucial factor for studying slow processes such as crystal nucleation and glassy dynamics.

I. INTRODUCTION

Hard spheres are perhaps the most commonly studied model system in soft matter physics. In addition to being excellent model systems for certain colloidal suspensions [1–3], their simplicity makes them a fundamental model study for many-body systems at any scale. As a result, simulations of hard spheres have been instrumental in shedding light on many aspects of statistical physics, including phase behavior in bulk[4–11] and in confinement [12–18], crystal nucleation[19–26], and glassy materials [27–36].

Due to their popularity as a model system, it should come as no surprise that significant efforts have been made to design efficient hard-sphere simulations, typically based on either Monte Carlo (MC) methods or event-driven molecular dynamics (EDMD). Monte Carlo methods have been applied to hard-sphere systems since the 1950s [37, 38] and continue to be in active use. For a hard-sphere system in the canonical ensemble, the basic MC algorithm consists of simple single-particle displacement moves, which are accepted as long as no overlaps are created [39]. This approach can readily be adapted to a variety of thermodynamic ensembles, or to incorporate biasing potentials or external fields, and scales well to large systems. Moreover, intelligently chosen MC trial moves, such as cluster moves[40–42] or swap moves [43], can lead to significant speedups in sampling speed. In particular, event-chain algorithms, which move chains of particles collectively in a rejection-free move, are highly efficient for hard-sphere systems [44, 45]. Depending on the goal of the simulations, one potential downside of MC simulations is the fact that, unless special care is taken[46, 47], the dynamics do not typically correspond well to that of a real-world system.

Event-driven molecular dynamics (EDMD) simulations are a second widely used method for simulating hard-sphere systems, with a similarly long history [48]. These simulations solve for the motion of the particles using Newton's equations of motion, effectively modeling the movement of hard spheres in a vacuum[1]. Following Newton's third law, hard spheres move in simple straight lines until they collide with a neighboring particle. EDMD simulations rely on predicting future collision events between particles, and then resolving each of these collision events in order. In comparison to standard MC simulations, hard-sphere EDMD simulations are extremely efficient, rivaling e.g. event-chain methods [45], with the added benefit of reproducing physically realistic dynamics. This is ideal for the study of glassy dynamics or spontaneous nucleation, where observing the dynamics of interest requires simulation over long time scales.

Here, we present a light-weight and fast hard-sphere EDMD code, which combines an efficient event-queueing system [51], neighbor lists [52], and "lazy" event evaluation [53]. We benchmark the simulation code on a range of monodisperse and bidisperse hard-sphere systems, and find significant performance gains when comparing the results with other commonly used simulation codes [54, 55].

In the remainder of this article, we discuss some of the more detailed aspects of the efficient implementation of EDMD simulations (Sec. II), show benchmarks of three variations of the presented simulation code (Sec. III), and discuss the results (Sec. IV).

II. IMPLEMENTATION

In this section, we outline the implementation of an efficient event-driven hard-sphere simulation algorithm. After briefly outlining the typical setup of a hard-sphere EDMD, we focus on three implementation choices: the event calendar, the neighbor selection, and the choice of how many events to schedule for a given particle.

For the simulations described in this article, we always consider periodic systems in an orthogonal simulation box.

[1] It should be noted that variations on EDMD that simulate Brownian[49] or Langevin[50] dynamics have been developed as well.

A. Basic outline

The basic premise of a hard-sphere EDMD is straightforward. Given a configuration of N particles with positions $\{\mathbf{r}^N\}$ and velocities $\{\mathbf{v}^N\}$, the algorithm determines which pair of particles will be the first to undergo a collision, and when this collision will take place. Once the event is determined, the simulation time t jumps forward to the collision time, particle positions and velocities are updated to reflect the impact of the collision, and the simulation proceeds to predict the next collision.

Since we are simulating purely hard particles moving in a vacuum, we know that in between collision events the particles experience no forces, and hence move in straight lines. As a result, for hard spheres, predicting the collision time for a pair of particles can be done analytically based on the initial distance vector $\mathbf{r}_{ij} = \mathbf{r}_j(t) - \mathbf{r}_i(t)$ and the relative velocity $\mathbf{v}_{ij} = \mathbf{v}_j(t) - \mathbf{v}_i(t)$ between the colliding particles i and j. Specifically, the time until the next collision $t_{\rm col}$ is given by [55]

$$t_{\rm col} = \frac{-b - \sqrt{b^2 - v_{ij}^2(r_{ij}^2 - \sigma_{ij}^2)}}{v_{ij}^2}, \qquad (1)$$

where $b = \mathbf{r}_{ij} \cdot \mathbf{v}_{ij}$ and the minimum approach distance $\sigma_{ij} = R_i + R_j$ is the sum of the radii R_i, R_j of the two spheres. Note that a collision is only expected to occur when $b < 0$ (i.e. the particles are currently moving towards each other), and the discriminant $b^2 - v_{ij}^2(r_{ij}^2 - \sigma_{ij}^2) \geq 0$.

When a collision occurs, the two particles involved will instantaneously undergo a change in their velocities. Assuming a perfectly elastic collision, this change conserves the total momentum and kinetic energy of the particles, resulting in velocity changes:

$$\delta\mathbf{v}_i = -\frac{2m_j}{m_i + m_j}\frac{b\hat{\mathbf{r}}_{ij}}{\sigma_{ij}}, \qquad \delta\mathbf{v}_j = \frac{2m_i}{m_i + m_j}\frac{b\hat{\mathbf{r}}_{ij}}{\sigma_{ij}^2}, \quad (2)$$

where m_i is the mass of particle i, $\hat{\mathbf{r}}_{ij}$ is a unit vector in the direction of $\mathbf{r}_{ij}$, and the values of b and $\hat{\mathbf{r}}_{ij}$ are taken at the moment of collision.

Using these equations for predicting and resolving collisions, a naive implementation of the algorithm could simply proceed as follows:

1. Check *all* particle pairs for possible collisions, and record the first collision time $t_{\rm first}$.
2. Set $t = t_{\rm first}$ and update *all* particle positions.
3. Process the collision by updating the velocities of the two colliding particles
4. Go to step 1.

In practice, this is rather inefficient, and there are a number of standard optimizations that can be implemented to improve performance. First, after a collision between particles i and j, any other collisions previously predicted for particle i and j are no longer valid, and hence the future collisions for these particles need to be recalculated. However, all predicted collisions that only involve particles *other* than i and j are still valid, since the trajectories of those particles did not change. To make use of this, we keep track of an *event calendar*: a list of predicted future events. Whenever a collision occurs, we then remove any future events that involve the colliding particles from the event calendar, but leave the others in place, avoiding the need to re-predict them. We will discuss the implementation of the event calendar in Sec. II B.

A second useful optimization lies in the fact the positions of any other particles are irrelevant for handling a collision between two particles. Hence, there is no need to update particle positions for any collision they are not involved in. Instead, for each particle i we can simply keep track of the last time t_i that the particle was updated, and its position and velocity at that time, and update its position only when needed. This avoids the vast majority of particle updates in the algorithm described above.

Finally, we can save additional time by realizing that particles can only collide with particles in their immediate environment. Hence, predicting collisions between particles that are separated by distances many times larger than the particle size is rather wasteful: most likely these events will be invalidated by intervening collisions long before they would take place. Efficiently dealing with particles with a finite interaction range is a common problem in computer simulations, and the standard approach for addressing this is to only consider a limited selection of nearby neighbors when calculating interactions or predicting collisions [39]. In EDMD simulations, the most commonly used method is the use of cell lists [54, 55], but neighbor list (also called Verlet lists) have also been implemented [52]. In both cases, we introduce a new event type, a neighborhood event, which triggers an update of the neighbor list or cell list position for a given particle. These methods will be discussed in more detail in Section II C.

As a result, the basic outline of a modern EDMD simulation is as follows.

1. Initialize the cell list or neighbor lists.
2. For each particle, predict the point in time when either it moves to a different cell or its neighbor list needs an update, and add this event to the event calendar.
3. For all neighboring particle pairs, predict future collisions and add them to the event calendar.
4. Locate the first collision or neighborhood event in the event calendar, and move the simulation time t forward to the associated event time.
5. If the current event is a neighborhood event, do the necessary bookkeeping on the particle environment, and predict new neighborhood and collision events for the updated particle.

6. If the current event is a collision, update the two colliding particles to time t, and process the collisions. Then, predict new neighborhood and collision events for both particles.

7. Go to step 4.

In the following subsections, we look at some of the implementation details in more detail.

B. Event calendar

The implementation of the event calendar in EDMD simulations has been discussed extensively in literature [51, 53, 55–57]. We are storing a large number of events (scaling with the system size N), each containing (at least) information about the event type, the time at which the event will occur, and the particles involved. The data structure we choose for this has to allow for

1. rapidly finding the event that is first chronologically (i.e. with the lowest event time),

2. efficient addition of new events, and

3. efficient deletion of invalidated events.

The first requirement strongly suggests that the events in the event calendar should be kept sorted based on their event time. Using an unsorted event calendar, finding the first event would require a search through all stored events ($\mathcal{O}(N)$ operations), which would be expensive to perform for every collision. A sorted data structure avoids this issue by ensuring that the next event in line is always easy to find. In particular, event calendars in the form of a *binary search tree* are commonly used [55], but other variations of binary tree structures have been investigated as well [57]. Regardless of the exact implementation, in binary tree structures the total cost of operations for handling a collision (i.e. the combination of finding the first event, deleting the old events for the particles involved, and inserting the newly predicted collisions) scales as $\mathcal{O}(\log N)$ in system size. Since the number of collisions per simulation time unit in an EDMD trivially scales as $\mathcal{O}(N)$, this results in a total computational cost per time unit of the simulation of $\mathcal{O}(N \log N)$.

A more efficient event calendar strategy was introduced in Ref. 51, which proposed combining a binary tree structure for events in the near future with an array of linear lists for the remaining events. In particular, this strategy divides the simulation time into intervals of length Δt, and chooses to *only* store events within the current time interval in a sorted binary tree structure as described above. The remaining events are stored inside a set of "event buckets", with each bucket corresponding to an unsorted list of all events that occur within the same time interval $t \in [k\Delta t, (k+1)\Delta t)$ (with k an integer). At the end of the current time interval (i.e. when no more events remain in the binary tree), the events from the next event bucket are added to the binary tree, and the simulation continues. Since the events in each bucket are implemented as unsorted doubly linked lists, addition or removal of an event from the buckets can be done at a fixed cost of $\mathcal{O}(1)$. Moreover, the cost of updating the initial binary tree scales only as $\mathcal{O}(\log n)$, with n the typical number of events scheduled in a time interval Δt. By choosing $\Delta t \propto 1/N$, this number n can be chosen to be independent of N, resulting in a $\mathcal{O}(1)$ cost for event scheduling and an overall EDMD simulation performance of $\mathcal{O}(N)$ operations per simulation time unit. This yields a significant performance improvement over implementations that rely on a pure binary tree, especially for larger systems (i.e. $N \gtrsim 10^3$) [51].

Note that in this approach, care needs to be taken that a sufficiently large number of event buckets are available. The number of buckets should typically be on the order of $t_{\max}/\Delta t$, where $t_{\max}$ is the maximum expected difference between the current time and a predicted event time. In practice, in an EDMD simulation this maximum time is not bounded, as two particles that happen to move slowly or nearly parallel can experience a collision arbitrarily far in the future. However, as described in Ref. 51, this issue can be circumvented by the use of an "overflow" bucket which contains the (rare) events that are too far into the future. In all our simulation codes presented here, we follow the algorithm described in Ref. 51 for the event calendar, with one minor modification: we use a binary search tree rather than a complete binary tree [57] as our initial event tree. Since the scheduling of events into the (small) binary tree takes up only a small fraction of the overall simulation time, we do not expect this choice to make a significant difference.

C. Neighbor selection

The majority of the computational cost of an EDMD simulation is spent on the prediction and scheduling of collisions. Hence, reducing the number of collision partners to check for a given particle can drastically affect simulation times. Broadly, there are two main strategies for determining which particles to check.

Most commonly used are *cell lists*: the simulation box is divided up into a grid of cells that are sufficiently large to ensure that collisions can only occur between particles in adjacent cells (i.e. particles that are at most one cell away in along any of the three spatial dimensions). We can then limit ourselves to only predicting collisions with particles that are either in the same cell as the particle being checked, or in any of the 26 (in 3D) adjacent cells. Since the number of particles per cell is limited, finding the future collisions for a given particles then only requires checking a limited number of neighbors, rather than all N particles in the system. This efficiency comes at the (small) cost of having to keep track of the cell each particle is in. To this end, an extra event type is

added that occurs when a particle crosses from one cell to the next. Like collisions, these cell-crossing events are predicted and added to the event calendar each time a particle is involved in an event. When a cell crossing occurs, particles in the newly adjacent cells (i.e. 9 cells in 3 dimensions) are checked for collisions.

Another common approach for neighbor selection is via the implementation of *neighbor lists* [39]. An efficient implementation of this approach for event-driven simulations was shown in Ref. 52. In this strategy, each particle maintains a list of particles that are considered to be "close enough" for possible interactions. In particular, we store for each particle i the center of its current neighborhood $\mathbf{r}_{\mathrm{neigh},i}$, and the set of other particles j such that

$$|\mathbf{r}_{\mathrm{neigh},i} - \mathbf{r}_{\mathrm{neigh},j}| < (1+\alpha)\sigma_{ij}, \tag{3}$$

with $\alpha > 1$ a parameter that controls the size of the neighborhood of a particle. We can then conclude that particle i cannot collide with any particle k *not* in its neighborhood, unless either i has moved by at least αR_i or k has moved by at least αR_k from their respective neighborhood centers. Hence, we can safely limit ourselves to predicting collisions only between particles that are considered to be neighbors, as long as for each particle, we update its neighbor list when it reaches a distance of α times its radius from the center of its environment.

In addition to particle collisions, we then also include an event for each particle that takes place when

$$|\mathbf{r}_i - \mathbf{r}_{\mathrm{neigh},i}| = \alpha R_i. \tag{4}$$

The prediction of this event is analogous to a normal sphere-sphere collision, with an event time

$$t_{\mathrm{neigh}} = \frac{-b_{n,i} + \sqrt{b_{n,i}^2 - v_i^2(|\mathbf{r}_i - \mathbf{r}_{\mathrm{neigh},i}|^2 - \alpha^2 R_i^2)}}{v_i^2}, \tag{5}$$

with $b_{n,i} = \mathbf{r}_i \cdot \mathbf{r}_{\mathrm{neigh},i}$. The neighbor list update affects not only the list of neighbors of particle i, but will also cause i to be added to (or removed from) the neighbor lists of nearby particles. Note that a cell list is still used to find possible neighbors of a given particle, in order to limit the number of distances that need to be checked. The cell size in this case is chosen to be large enough to ensure that only particles in neighboring cells can be in each others neighbor list. Updates to the cell list are done as part of the neighbor list update.

Analogous to Ref. [52], we implement the neighbor list update of particle i as follows:

1. Update particle i to the current simulation time.
2. If particle i has left the simulation box, apply periodic boundaries.
3. Set $\mathbf{r}_{\mathrm{neigh},i} = \mathbf{r}_i$
4. Update the cell list with the new particle location.
5. For all particles j that were previously neighbors of i, remove i from the neighbor list of j and *vice versa*.
6. Check all 27 neighboring cells for particles j that satisfy: $|\mathbf{r}_{\mathrm{neigh},j} - \mathbf{r}_{\mathrm{neigh},i}| < (1+\alpha)\sigma_{ij}$. For each of these, add i to the neighbor list of j and *vice versa*.
7. Delete any future events for particle i from the event calendar.
8. Predict and schedule new collision and neighbor list events for particle i.

We implement the neighbor list as a fixed-length array for each particle, allowing a maximum number $n_{\mathrm{neigh}}^{\mathrm{max}}$ of neighbors that is constant during the simulation and should be chosen large enough to accommodate the maximum number of neighbors that a particle can have at the chosen shell size α. For the hard-sphere systems studied here, we found that a shell size $\alpha = 0.5$ gave near optimal results (see SI). For this choice, $n_{\mathrm{neigh}}^{\mathrm{max}} = 24$ was sufficient in all cases [2].

It should be emphasized that the cell list and neighbor list approaches will perform optimally under different circumstances. In particular, neighbor lists have excellent performance in systems where particles are likely to remain in the same environment for a long time, as is the case in solid phases and glassy fluids. Under these circumstances, the neighbor lists require few updates, and the number of calculated collisions is typically lower than in a cell list approach. For example, in a monodisperse hard-sphere crystal at a number density $\rho\sigma^3 \approx 1$, the neighbor list for each particle is likely to only contain its 12 nearest neighbors, while checking 27 cells of a size of at least σ^3 each requires checking on the order of 27 particles. In contrast, in low-density fluids, the high particle mobility leads to much more frequent neighbor list updates. As neighbor lists updates are more expensive than cell list updates, this favors the cell list method. Since long simulation times are more likely to be a concern in systems with relatively slow dynamics (e.g. glassy systems), we focus here on the neighbor list approach, but include benchmarks for one code (labeled **CellList**) using cell lists.

D. Lazy event scheduling

A key point of consideration in the implementation of an EDMD simulation code is the question of how

[2] Note that in principle, it is possible to implement a variation on this approach where the shell size of a specific particle is temporarily reduced if too many neighbors are encountered [52], but this was found to be unnecessary for the systems considered here, where a good upper bound on the maximum number of nearest neighbors can be estimated.

many events to schedule per particle. In the above, we have implied that all predicted collisions (and the neighbor list update) for each particle are added to the event tree, but this is not necessarily the most efficient choice. Lubachevsky[53] noted that when predicting many events per particle, the later events are likely to be pre-empted by the earlier ones, and hence the computational cost of adding them to the event calendar might well be a waste. As a result, many studies (e.g. Refs. 52, 53, and 58) choose to schedule only a single event per particle. If the event scheduled for particle i is invalidated by a collision between two other particles, the event is typically converted into an update event, which simply rechecks for future collisions and neighborhood events without changing the particle trajectory. Although this means that sometimes the same collision is predicted more than once, the reduction in computation cost due to scheduling fewer events can result in significant speedups [58]. An alternative strategy, used in e.g. Refs. [54, 56] does maintain all predicted events per particle, but still schedules only one event per particle in the event tree (limiting its size), while separately maintaining lists of sorted events per particle.

Here, we explore both the option of scheduling multiple events per particle and scheduling only a single event per particle. In particular, we consider two EDMD codes, both using neighbor lists and the event scheduling approach from Ref. 51, with the first (labeled **MultiEvent**) scheduling multiple events per particle, and the second (labeled **SingleEvent**) scheduling only one event per particle.

Specifically, in the **MultiEvent** code, we schedule all collision events that take place before the next neighbor list update event for both colliding particles. Since our implementation of the neighbor list update of particle i completely reevaluates the future events of i, there is no benefit to scheduling any events beyond this time.

In contrast, in the **SingleEvent** code, we only schedule the first (collision or neighbor list update) event for each particle. In order to check for invalidated events, each particle keeps track of the total number of collisions $n_{\rm col}$ it has experienced. In particular, when updating the event for particle i (i.e. after a collision or neighbor list update), we do the following:

1. Predict the new neighbor list update for particle i, and set $t_{\rm min}$ equal to the predicted time.

2. For all neighbors of i, predict a possible future collision. If a predicted collision time is smaller than $t_{\rm min}$, set $t_{\rm min}$ equal to the collision time, and remember the collision partner.

3. Schedule the new (neighbor list or collision) event for particle i at $t_{\rm min}$.

4. If the predicted event was a collision (with particle j), store $n_{{\rm col},j}$ as additional information with the event.

Then, when handling the collision that was predicted for particle i with collision partner j, we:

1. Remove the event from the event calendar.

2. Check that the value of $n_{{\rm col},j}$ stored with the event predicted for particle i is equal to the current value of $n_{{\rm col},j}$.

3. If not, then the collision is no longer valid. Predict the next event for particle i, and move on to the next event (ignore the steps below).

4. Otherwise, resolve the collision between the two particles, and increment $n_{{\rm col},i}$ and $n_{{\rm col},j}$.

5. Remove the scheduled event for j from the event calendar, and predict new events for both particles.

Our approach is similar to the one followed in Ref. 52, with the distinction that we make no attempt to ensure that collision predictions are symmetric between collision partners. For example, if the first collision found for particle i is with particle j, then this does not affect the predicted collision for particle j, regardless of the predicted collision time. Since the $i-j$ collision will be handled when the event associated with particle i is handled, there is no need to also schedule the same event in reverse. Moreover, the original prediction for j may still take place if the $i-j$ collision is invalidated, so it is beneficial to leave the original event prediction for j in place.

In addition to saving significant amounts of scheduling effort and memory usage, this scheduling approach allows for a few small optimizations in the collision prediction. First, the strict time limit on predicted events can sometimes avoid the full calculation of the collision time in Eq. 1, by noting that $t_{\rm col}$ is always greater than $(r_{ij}^2-\sigma_{ij}^2)/2b$. Hence, if this lower bound is already greater than the current $t_{\rm min}$, there is no need to calculate the exact collision time. Additionally, the total number of events that are scheduled is now equal to the number of particles, plus a small number of special events that are used for e.g. performing measurements, writing simulation output, or applying a thermostat. This eliminates the need for separate data structures for events and particles, allowing us to combine both into a single structure. This may help in optimizing memory access. Finally, the typically small times for predicted events means that the number of event buckets used in the event calendar can be quite limited, reducing memory usage.

III. BENCHMARKS

We test three versions of our simulation code (**CellList**, **MultiEvent**, and **SingleEvent**) on both monodisperse and bidisperse hard-sphere fluids, and compare their performance to other available EDMD codes [54,

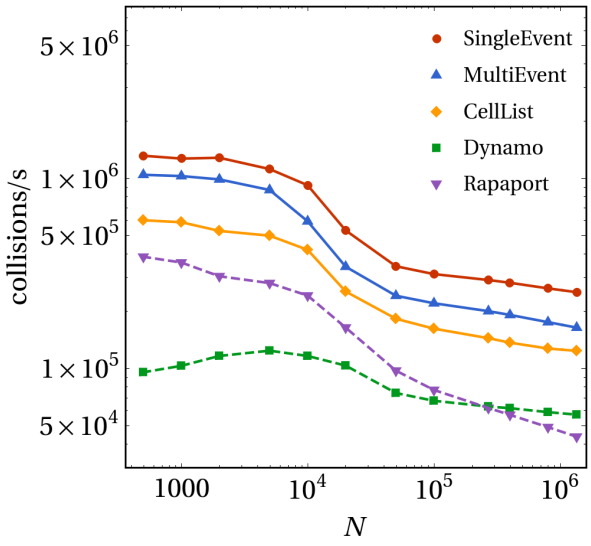

FIG. 1. Benchmark results on monodisperse hard-sphere fluids for different simulation codes and a range of system sizes N. The label Rapaport indicates the code provided with Ref. [55]. The label Dynamo indicates the simulation package Dynamo [54].

55]. Some additional practical details of our simulation codes are described in the Supplemental Information (SI). Initial disordered configurations were created using a variation of the **SingleEvent** code where all particles grow in size during the simulation. This results in a rapid compression from a dilute random configuration to the desired packing fraction. After reaching the desired packing fraction, the system was then equilibrated using the **SingleEvent** code. In order to cut down on the computational cost of equilibration (which can be slow in the dense binary system), all initial configurations for systems larger than $N = 10^5$ particles were created by duplicating a smaller configuration either $2 \times 2 \times 2$ times or $3 \times 3 \times 3$ times.

For all benchmarks, the simulation is run for at least 10^7 collisions, and we measure the average number of collisions processed per second, starting the measurement after the initialization of the system. All three simulation codes are implemented in C. Compilation and optimization details are given in the SI. All benchmarks were performed on a workstation containing two 3.3GHz Intel Xeon Gold 6234 CPUs, with 8 CPU cores each, using Ubuntu 20.04 as the operating system . For each benchmark, both CPUs were filled by running 16 identical jobs in parallel. Hyperthreading was disabled.

A. Monodisperse hard spheres

First, we simulate a monodisperse hard-sphere fluid at packing fraction $\eta = 0.49$, for a range of system sizes N ranging from 500 to 1.35×10^6 particles. This packing fraction is slightly below the freezing packing fraction [39]. We compare all three codes to both the code by Rapaport, provided with Ref. [55][3], and the simulation package Dynamo developed by Bannerman *et al.*[54], version 1.7.6.

In Fig. 1, we show the number of collisions per second processed by each simulation. We observe a significant variation in simulation speeds, which is largely consistent as a function of system size. It is interesting to note that even for this relatively mobile hard-sphere fluid, the introduction of a neighbor list (i.e. switching from the **CellList** to **MultiEvent** code) results in a speedup of around 50% accross all system sizes. Combining this with the choice to store only one event per particle (i.e. the **SingleEvent** code) provides another similar speedup. In total, the **SingleEvent** code outperforms Dynamo by a factor 4-10, depending on system size.

All benchmarked simulations show a noticeable decrease in efficiency around a system size of $\approx 10^4$, which we attribute to cache misses. The simulation methods examined here typically use a few hundred bytes of memory per particle, and the total cache size of the used processors is approximately 2.5MB per core. This means that for small systems (up to a few thousand particles), all data relevant to the simulation fits inside the processor's cache. For larger systems significant data exchange between CPU and the main memory is needed, which drastically slows down performance. As some of the CPU cache is shared between cores, this performance drop can be shifted to slightly larger system sizes by running only a single job at a time (see SI). Note that in EDMD simulations, particle data is typically accessed in an essentially random order: two consecutive collisions can take place anywhere in the system, and particles that are close together in space may not be stored close together in memory. As a result, optimizing EDMD simulations with respect to cache usage is inherently difficult.

As Fig. 1 shows, the Rapaport code shows a stronger decrease in performance for large systems than the other methods. This is due to its use of a $\mathcal{O}(\log N)$ binary search tree as the event calendar, in contrast to the $\mathcal{O}(1)$ scheduling methods used by the other codes.

B. Bidisperse hard spheres

As our second test system, we take a binary hard-sphere system with a size ratio $q = 0.85$ and a composition $x_L = N_L/N = 0.3$ (where N_L is the number of large spheres in the system). This system has been explored in several studies of glassy dynamics [34–36]. Note that since the Rapaport code[55] implements only monodisperse systems, we did not include it in the comparison here.

[3] Note that small modifications to the initialization and output of the code were made to allow the code to read in an initial configuration and report the elapsed time during the simulation.

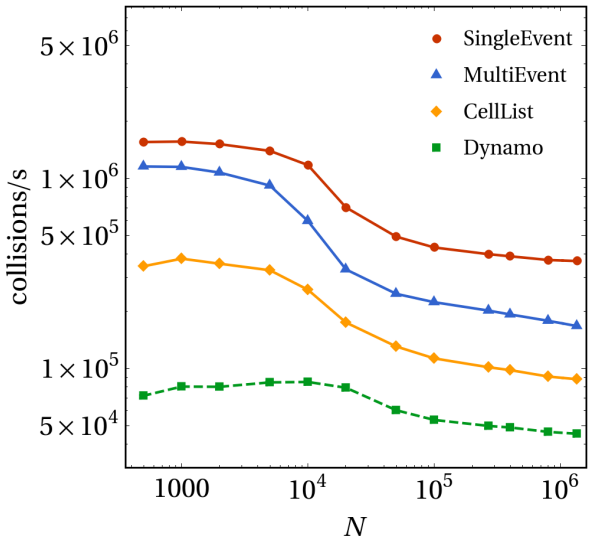

FIG. 2. Benchmark results on bidisperse hard-sphere fluids with size ratio $q = 0.85$, composition $x_L = 0.3$, and packing fraction $\eta = 0.58$, as a function of the system size N.

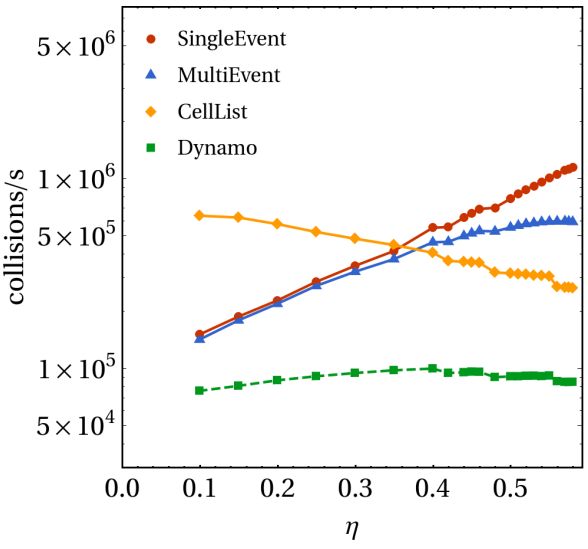

FIG. 3. Benchmark results on bidisperse hard-sphere fluids with size ratio $q = 0.85$, composition $x_L = 0.3$, and system size $N = 10^4$, as a function of the packing fraction η.

For this binary mixture, we first benchmark the simulation performance as a function of system size at a fixed packing fraction $\eta = 0.58$. The results are shown in Fig. 2. We observe similar results as for the monodisperse system, with again significant performance gaps between the different simulation codes. In this system, dynamics are glassy: particles are trapped by their neighbors for significant amounts of time before escaping their cage. As a result, neighbor updates are relatively rare, and hence the effect of incorporating a neighbor list is more significant than in Fig. 1: it results in a speed-up of approximately a factor two. Additionally, the typical number of events predicted per particle in the **MultiEvent** code is higher in this system than in the lower-density monodisperse system, resulting in an increased performance boost from switching to the **SingleEvent** code.

In order to investigate the effect of packing fraction on performance, we also benchmark each simulation code at a fixed system size of $N = 10^4$ particles for a range of different packing fractions. As shown in Fig. 3, we find a significant variation in the efficiency of the different codes as the packing fraction changes. As already hinted at in Sec. II C, the **CellList** code performs excellently at low packing fractions, but slows down as the packing fraction increases. This is understandable: as the packing fraction increases, more and more particles have to be checked for collisions. As most of these events will be pre-empted by an earlier collision, many of these calculations are wasted, resulting in an overall slowdown with increasing packing fraction. The step-wise performance decreases that can be seen for both the **CellList** code and Dynamo in Fig. 3 are due to the discrete reduction the number of cells that fit in the simulation box. A step occurs at a packing fraction where the number of cells is reduced by 1, which results in a discontinuous increase in the average number of particles per cell, and hence an increase in the computation cost of checking for collisions. Similar steps can be seen even for the **MultiEvent** and **SingleEvent** codes, but only at lower packing fractions ($\eta \lesssim 0.5$) where neighbor list updates (which rely on the cell list) account for a significant part of the simulation time.

In contrast to the **CellList** code, in terms of collisions per second, the neighbor-list based codes become *faster* with increasing packing fraction. As the packing fraction increases, particle mobility decreases and hence fewer neighbor list updates are required[4]. As a result, both neighbor-list driven codes outperform the **CellList** code at packing fractions $\eta \gtrsim 0.4$. Moreover, at higher packing fractions the number of unnecessary events predicted scheduled per particle in the **MultiEvent** code is larger, and hence the **SingleEvent** code becomes the fastest option by a significant margin. Note that most hard-sphere simulation studies where long simulations are required are likely to focus on the high-packing fraction fluid regime, where e.g. glassy behavior or spontaneous crystal nucleation can be observed.

[4] Note that this effect does not hold for the cell list updates in the **CellList** code: for a homogeneous system, the average (bidirectional) flux of particles through any cell wall is only a function of the particle density and the typical speed (i.e. temperature) of the particles. While some particles may be trapped for a long time in a single cell, this is compensated by other particles trapped vibrating around the edge of a cell and hence undergoing many cell crossings.

IV. DISCUSSION AND CONCLUSIONS

The simulation codes presented here (and in particular the **SingleEvent** code) clearly perform well in comparison to other publicly available EDMD implementations. It should be noted that part of this efficiency is due to the fact that the code was written with only hard-sphere systems in mind. In contrast, Dynamo[54] offers the possibility to simulate different particle shapes, different boundary conditions, stepped pair interactions, as well as a variety of other features. However, for the common use case of studying hard-sphere systems in a periodic rectangular box, as is often necessary for e.g. glass or nucleation studies, the presented difference in simulation speed (typically a factor 5 to 10) may be a crucial factor, saving significantly on both CPU time and energy consumption.

It should be noted that if the goal is simply to equilibrate a hard-sphere system or to sample equilibrium states, it is not *a priori* clear that EDMD simulations are the ideal choice. Recent work by Klement *et al.* demonstrated that hard-sphere systems can be highly efficiently equilibrated using Newtonian Event Chains (NEC) [45], an approach that blends aspects of Monte Carlo simulations and event-driven dynamics. Simulating a monodisperse system of hard spheres at $\eta = 0.49$, they observed that their NEC simulations were significantly faster than EDMD simulations in terms of particle displacements per hour, which directly translated into a similar speedup in terms of the simulation time required to simulate e.g. particle diffusion or crystal melting. In Table I, we compare the number of displacements per second achieved by the currently available implementation of NEC [59] in HOOMD[60] to the number of collisions per second of the **SingleEvent** code for several system sizes. We see that a sufficiently efficient EDMD code can indeed slightly outperform this implementation of NEC [5]. However, it should be noted that since the complexity of an NEC displacement is lower than that of an EDMD collision, further optimization of the NEC implementation may be possible. Moreover, as also noted in Ref. 45, the local nature of the NEC algorithm allows for more straightforward parallelization [61]. On the other hand, the EDMD approach has the significant advantage of producing physically realistic dynamics.

N	NEC displacements/s	EDMD collisions/s
1 000	1.0×10^6	1.6×10^6
10 000	8.8×10^5	1.2×10^6
100 000	4.1×10^5	4.3×10^5
1 350 000	2.2×10^5	3.7×10^5

TABLE I. Comparison between Newtonian Event Chain (NEC) simulations [45, 59], and the **SingleEvent** EDMD code, for a selection of the systems shown in Fig. 2. Following Ref. 45, we compare the number NEC particle displacements per second to EDMD collisions per second.

Although here we focus purely on single-processor simulation codes, simulations of large systems can often be parallelized to decrease simulation time, although this typically carries an overhead cost in total CPU time. For Monte Carlo methods, highly parallelized approaches have been shown to be capable of extremely fast equilibration of 2D hard-disk systems [62, 63]. A similar approach may be effective for 3D hard spheres as well. In contrast, EDMD simulations are notoriously difficult to parallelize [64], due to the fact that two events that are spatially far apart could influence each other even if they occur at nearly the same time, as long as they are connected by a chain of (nearly) touching particles. Nonetheless, several effective methods for parallelization of EDMD simulations have been demonstrated (see e.g. [58, 64]), and can in principle be extended to the EDMD codes presented here.

As a final note, it is interesting that in all of the benchmarks, we found that the **SingleEvent** code performed at least as efficiently as the **MultiEvent** code. This observation is likely to be general for hard spheres and any extensions of the code to other spherically symmetric systems. In these systems, the cost of predicting a collision is low, and hence the cost of having to sometimes recalculate the same prediction is acceptable. However, it is worth noting that in EDMD simulations of anisotropic particles, such as polyhedra, ellipsoids, or patchy particles [52, 65–69], the computational cost of predicting a collision event is typically much higher, since collisions for these particles cannot be predicted analytically. This may favor an approach where multiple events are stored per particle, to avoid repeated predictions of the same event. On the other hand, the restriction to a single event per particle also helps to limit the time window to search for possible collisions, hence providing an efficient "early out" for numerical collision-prediction routines. Hence, the benefit of single-event scheduling approaches may depend on the system under consideration.

In summary, we have presented a set of light-weight EDMD simulation codes for hard-sphere systems that appear to perform very efficiently in comparison to the the simulations currently used in literature on e.g. the glassy dynamics of hard spheres. Simulating these systems more efficiently allows for a significant saving of CPU time and energy.

The simulation codes are freely available at:
https://github.com/FSmallenburg/EDMD

[5] It should be noted that the number of NEC displacements per second in Table I is significantly lower than those reported in Ref. 45, especially for larger systems. We attribute this to our protocol of filling up the CPU with an identical number of jobs when performing benchmarks. When running only a single job per CPU, we observe performance in line with Ref. 45, with EDMD still outperforming NEC in terms of displacements per second.

V. ACKNOWLEDGEMENTS

I would like to thank Michael Engel, Marco Klement, and Joshua Anderson for helpful discussions and aid in the comparison to the Newtonian Event Chain algorithm, and Laura Filion for many useful discussions.

VI. REFERENCES

[1] P. N. Pusey and W. Van Megen, Nature **320**, 340 (1986).
[2] W. K. Kegel and A. van Blaaderen, Science **287**, 290 (2000).
[3] C. P. Royall, W. C. Poon, and E. R. Weeks, Soft Matter **9**, 17 (2013).
[4] D. Frenkel and A. J. Ladd, The Journal of chemical physics **81**, 3188 (1984).
[5] T. Biben and J.-P. Hansen, Physical review letters **66**, 2215 (1991).
[6] M. Dijkstra, R. van Roij, and R. Evans, Physical review letters **81**, 2268 (1998).
[7] M. Dijkstra, Physical Review E **58**, 7523 (1998).
[8] M. Dijkstra, R. van Roij, and R. Evans, Physical Review E **59**, 5744 (1999).
[9] L. Filion, M. Hermes, R. Ni, E. Vermolen, A. Kuijk, C. Christova, J. Stiefelhagen, T. Vissers, A. Van Blaaderen, and M. Dijkstra, Physical review letters **107**, 168302 (2011).
[10] P. K. Bommineni, N. R. Varela-Rosales, M. Klement, and M. Engel, Physical review letters **122**, 128005 (2019).
[11] P. K. Bommineni, M. Klement, and M. Engel, Physical Review Letters **124**, 218003 (2020).
[12] M. Schmidt and H. Löwen, Physical Review E **55**, 7228 (1997).
[13] A. Fortini and M. Dijkstra, Journal of Physics: Condensed Matter **18**, L371 (2006).
[14] M. Gordillo, B. Martínez-Haya, and J. M. Romero-Enrique, The Journal of chemical physics **125**, 144702 (2006).
[15] E. C. Oğuz, M. Marechal, F. Ramiro-Manzano, I. Rodriguez, R. Messina, F. J. Meseguer, and H. Löwen, Physical review letters **109**, 218301 (2012).
[16] B. De Nijs, S. Dussi, F. Smallenburg, J. D. Meeldijk, D. J. Groenendijk, L. Filion, A. Imhof, A. Van Blaaderen, and M. Dijkstra, Nature materials **14**, 56 (2015).
[17] G. Jung and C. F. Petersen, Physical Review Research **2**, 033207 (2020).
[18] D. Wang, T. Dasgupta, E. B. van der Wee, D. Zanaga, T. Altantzis, Y. Wu, G. M. Coli, C. B. Murray, S. Bals, M. Dijkstra, *et al.*, Nature Physics **17**, 128 (2021).
[19] S. Auer and D. Frenkel, Nature **409**, 1020 (2001).
[20] S. Punnathanam and P. Monson, The Journal of chemical physics **125**, 024508 (2006).
[21] L. Filion, M. Hermes, R. Ni, and M. Dijkstra, The Journal of chemical physics **133**, 244115 (2010).
[22] R. Ni, F. Smallenburg, L. Filion, and M. Dijkstra, Molecular Physics **109**, 1213 (2011).
[23] J. T. Berryman, M. Anwar, S. Dorosz, and T. Schilling, The Journal of chemical physics **145**, 211901 (2016).
[24] D. Richard and T. Speck, The Journal of chemical physics **148**, 124110 (2018).
[25] N. Wood, J. Russo, F. Turci, and C. P. Royall, The Journal of chemical physics **149**, 204506 (2018).
[26] J. R. Espinosa, C. Vega, C. Valeriani, D. Frenkel, and E. Sanz, Soft Matter **15**, 9625 (2019).
[27] M. Rintoul and S. Torquato, The Journal of chemical physics **105**, 9258 (1996).
[28] R. J. Speedy, Molecular Physics **95**, 169 (1998).
[29] M. Leocmach and H. Tanaka, Nature communications **3**, 1 (2012).
[30] E. Sanz, C. Valeriani, E. Zaccarelli, W. C. Poon, M. E. Cates, and P. N. Pusey, Proceedings of the National Academy of Sciences **111**, 75 (2014).
[31] P. Urbani and F. Zamponi, Physical review letters **118**, 038001 (2017).
[32] L. Berthier, P. Charbonneau, D. Coslovich, A. Ninarello, M. Ozawa, and S. Yaida, Proceedings of the National Academy of Sciences **114**, 11356 (2017).
[33] E. Lázaro-Lázaro, J. A. Perera-Burgos, P. Laermann, T. Sentjabrskaja, G. Pérez-Ángel, M. Laurati, S. U. Egelhaaf, M. Medina-Noyola, T. Voigtmann, R. Castañeda-Priego, *et al.*, Physical Review E **99**, 042603 (2019).
[34] S. Marín-Aguilar, H. H. Wensink, G. Foffi, and F. Smallenburg, Physical Review Letters **124**, 208005 (2020).
[35] E. Boattini, F. Smallenburg, and L. Filion, Phys. Rev. Lett. **127**, 088007 (2021).
[36] E. Boattini, S. Marín-Aguilar, S. Mitra, G. Foffi, F. Smallenburg, and L. Filion, Nature communications **11**, 1 (2020).
[37] M. N. Rosenbluth and A. W. Rosenbluth, The Journal of Chemical Physics **22**, 881 (1954).
[38] W. W. Wood and J. Jacobson, The Journal of Chemical Physics **27**, 1207 (1957).
[39] D. Frenkel and B. Smit, *Understanding Molecular Simulations: From Algorithms to Applications* (Academic Press, San Diego, 2002).
[40] C. Dress and W. Krauth, Journal of Physics A: Mathematical and General **28**, L597 (1995).
[41] N. G. Almarza, The Journal of chemical physics **130**, 184106 (2009).
[42] D. J. Ashton, N. B. Wilding, R. Roth, and R. Evans, Physical Review E **84**, 061136 (2011).
[43] L. Berthier, D. Coslovich, A. Ninarello, and M. Ozawa, Physical review letters **116**, 238002 (2016).
[44] E. P. Bernard, W. Krauth, and D. B. Wilson, Physical Review E **80**, 056704 (2009).
[45] M. Klement and M. Engel, The Journal of chemical physics **150**, 174108 (2019).
[46] E. Sanz and D. Marenduzzo, The Journal of chemical physics **132**, 194102 (2010).
[47] A. Cuetos and A. Patti, Physical Review E **92**, 022302 (2015).
[48] B. J. Alder and T. E. Wainwright, The Journal of chemical physics **27**, 1208 (1957).
[49] A. Scala, T. Voigtmann, and C. De Michele, The Journal of chemical physics **126**, 134109 (2007).
[50] A. Scala, Physical Review E **86**, 026709 (2012).
[51] G. Paul, Journal of Computational Physics **221**, 615 (2007).
[52] A. Donev, S. Torquato, and F. H. Stillinger, Journal of computational physics **202**, 737 (2005).
[53] B. D. Lubachevsky, Journal of Computational Physics **94**, 255 (1991).

[54] M. N. Bannerman, R. Sargant, and L. Lue, Journal of computational chemistry **32**, 3329 (2011).
[55] D. C. Rapaport and D. C. R. Rapaport, *The art of molecular dynamics simulation* (Cambridge university press, 2004).
[56] M. Marın, D. Risso, and P. Cordero, Journal of Computational Physics **109**, 306 (1993).
[57] M. Marín and P. Cordero, Computer Physics Communications **92**, 214 (1995).
[58] M. A. Khan and M. C. Herbordt, Journal of computational physics **230**, 6563 (2011).
[59] "https://github.com/glotzerlab/hoomd-blue/tree/newtonian_event_chain_monte_carlo," [Accessed: 14-10-2021].
[60] J. A. Anderson, J. Glaser, and S. C. Glotzer, Computational Materials Science **173**, 109363 (2020).
[61] M. Klement, S. Lee, J. A. Anderson, and M. Engel, arXiv preprint arXiv:2104.06829 (2021).
[62] J. A. Anderson, E. Jankowski, T. L. Grubb, M. Engel, and S. C. Glotzer, Journal of Computational Physics **254**, 27 (2013).
[63] M. Engel, J. A. Anderson, S. C. Glotzer, M. Isobe, E. P. Bernard, and W. Krauth, Physical Review E **87**, 042134 (2013).
[64] S. Miller and S. Luding, Journal of Computational Physics **193**, 306 (2004).
[65] A. Donev, S. Torquato, and F. H. Stillinger, Journal of computational physics **202**, 765 (2005).
[66] L. Hernández de la Peña, R. van Zon, J. Schofield, and S. B. Opps, The Journal of chemical physics **126**, 074105 (2007).
[67] F. Smallenburg, L. Filion, M. Marechal, and M. Dijkstra, Proceedings of the National Academy of Sciences **109**, 17886 (2012).
[68] F. Smallenburg and F. Sciortino, Nature Physics **9**, 554 (2013).
[69] S. Marín-Aguilar, H. H. Wensink, G. Foffi, and F. Smallenburg, Soft matter **15**, 9886 (2019).