Computer Physics Communications 229 (2018) 148–161

Contents lists available at ScienceDirect

Computer Physics Communications

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

GPU parallel simulation algorithm of Brownian particles with excluded volume using Delaunay triangulations

Francisco Carter [a,*], Nancy Hitschfeld [a], Cristóbal A. Navarro [b], Rodrigo Soto [c]

[a] Department of Computer Science, FCFM, Universidad de Chile, Santiago, Chile
[b] Institute of Informatics, Universidad Austral de Chile, Valdivia, Chile
[c] Physics Department, FCFM, Universidad de Chile, Santiago, Chile

ARTICLE INFO

Article history:
Received 10 March 2017
Received in revised form 4 February 2018
Accepted 10 April 2018
Available online 17 April 2018

Keywords:
Parallel computing
Particle dynamics
Brownian dynamics
Overlap correction
Delaunay triangulations
CUDA
GPGPU
N-body simulation

ABSTRACT

A novel parallel simulation algorithm on the GPU, implemented in CUDA and C++, is presented for the simulation of Brownian particles that display excluded volume repulsion and interact with long and short range forces. When an explicit Euler–Maruyama integration step is performed to take into account the pairwise forces and Brownian motion, particle overlaps can appear. The excluded volume property brings up the need for correcting these overlaps as they happen, since predicting them is not feasible due to the random displacement of Brownian particles. The proposed solution handles, at each time step, a Delaunay triangulation of the particle positions because it allows us to efficiently solve overlaps between particles by checking just their neighborhood. The algorithm starts by generating a periodic Delaunay triangulation of the particle initial positions on CPU, but after that the triangulation is always kept on GPU memory. We used a parallel edge-flip implementation to keep the triangulation updated during each time step, checking previously that the triangulation was not rendered invalid due to the particle displacements. We designed and implemented an exact long range force simulation with an all-pairs N-body simulation, tiling the particle interaction computations based on the warp size of the target device architecture. The resulting implementation was validated with two models of active colloidal particles, also showing a speedup of up to two orders of magnitude when compared to a sequential implementation. A short range forces simulation using Verlet lists for neighborhood handling was also developed and validated, showing similar performance improvements.

1. Introduction

A colloidal suspension is a mixture of microscopical insoluble particles dispersed throughout a continuous fluid, where particle sizes range from 1 nm to 10 μm. Colloidal suspensions appear in several natural and artificial substances as the milk, mud, inks, cosmetics or latex paint, for example. Also, they are used in many intermediate industrial processes. The interactions between colloidal particles of various kinds [1] have effects on the physical and chemical properties of the mixture such as its viscosity or light dispersion. To study these and other properties it is necessary to simulate particle systems of growing numbers ($N \geq 10^4$). Also, colloids are being used as models for active systems, to describe the motion of self-propelled microorganisms [2–4].

Colloids can be modeled as hard bodies subject to Brownian diffusive motion. Colloidal particles can typically interact through the fluid in what is called hydrodynamic interactions, via electrostatic forces for charged colloids, which can be screened in an electrolyte, or with van der Waals forces [1]. In out of equilibrium conditions, phoretic forces also appear [5]. Except for the hydrodynamic forces, these interactions can be modeled with good approximation as pairwise additive forces, which in out of equilibrium conditions can eventually break the action–reaction symmetry.

The simulation of colloidal dispersions can be divided on two main problems or phases executed in sequence. The first phase consists in updating the positions of the particles due to the interparticle interactions, according to some defined integration rule. The second task corresponds to ensure that the bodies do not overlap after the movement produced by the previous phase, in order to respect the excluded volume interaction for all involved particles. These problems are specific instances of the n-body simulation and collision detection respectively [6]. In some contexts, the simulation of colloidal particles is referred as Brownian dynamics.

There are two main methods for solving overlaps between particles: correcting all of them at once after they happen or use an event-driven approach, integrating the system until the collision instant, process the involved particles and repeat until the system

* Corresponding author.
E-mail address: francisco.carter@ug.uchile.cl (F. Carter).

https://doi.org/10.1016/j.cpc.2018.04.006
0010-4655/

reaches the target time step. The last method, is particularly useful when inertia is important and collisions result in rebounds as in granular materials [7,8]. It requires knowing the positions of the involved bodies at the time of collision, which becomes difficult when random Brownian motion is present. For the simulation of colloidal particles, which lack of inertia and excluded volume acts like a boundary condition rather than producing collisions, the first method is more suited.

This work focuses on designing and implementing a novel parallel simulation algorithm for 2D colloidal particle interacting with short and long range pairwise forces, with periodic boundaries, excluded volume and Brownian motion. The algorithm implementation takes advantage of the data-parallel computing capabilities of the GPU architecture, which have proven to be effective at accelerating the simulation process of several computational physics problems [9–11]. The interaction forces are allowed to be non-reciprocal as in the case of active particles [12,13]. The main contribution of this work consists of a new and efficient method of resolving particle overlaps by using periodic Delaunay triangulations, which are fully maintained on the graphic card. The starting positions and triangulation are initialized on the host while all the simulation code is executed on the device. The random values are also generated on the graphics card, both on the initialization and simulation phases. The algorithm uses a GPU edge-flip implementation to keep the triangulation fulfilling the Delaunay condition during each time step and to correct inverted triangles in case they are generated due to the particle displacements. For the short range force simulation, we developed a parallel algorithm that builds and uses Verlet lists in order to handle the particle neighborhood in parallel. The algorithm is validated with two models of active colloidal particles. Upon testing the parallel implementation of a long range forces simulation, the results show a performance improvement of up to two orders of magnitude when compared to the previously existing sequential solution. The algorithm for the short range force presents a similar performance improvement regarding the parallel long range implementation.

The paper is organized as follows: Section 2 describes the specific conditions and properties that the simulated systems must operate under. Section 3 lists previous related work used to solve similar problems. Section 4 details the designed solution with its subcomponents, data structures, and optimizations. The implementation of the algorithm is described in Section 5. Sections 6 and 7 cover the tests, benchmarks, validation and used methodology, presenting the running time and performance results when compared to the other implemented solutions. Finally, Section 8 rounds up the obtained results.

2. Description of the model

This section contains the description of the involved concepts and properties of this problem that may differentiate it from other body simulation problems, such as the excluded volume and stochastic component of particle movement.

2.1. Preliminaries

Let $P = \{p_1, p_2, \ldots, p_N\}$ be a set of N bodies on a d-dimensional space. The n-body simulation is the computation of the interactions over each body in P, where F_i corresponds to the interaction over p_i by effect of $P_i = P \setminus \{p_i\}$, the set of all other bodies in the system. The interaction forces F typically depend on the distance r_{ij} between two bodies as $F \sim r_{ij}^{-q}$. If $q > d$ the force is said to be short ranged, while if $q \leq d$, it is considered a long range force.

When the forces are long ranged, the set P_i cannot be reduced in n-body simulations and an exact evaluation of the forces has a cost $O(N^2)$. Approximate solutions for long range interactions as the Barnes–Hut algorithm reduce the cost to $O(N \log N)$ [14]. But for short range forces, the interactions can be truncated and, therefore, P_i can be reduced to the neighborhood of particles close to p_i. In this case, the evaluation of the forces costs $O(N * N_{NL})$ on average, where N_{NL} is the average number of neighbors of a body [15]. Since the construction of the list is $O(N^2)$ for evaluating all pairwise distances between bodies, it is possible to partition the simulation domain in cells so that close bodies get binned together in the same cells. Assignment of bodies to their respective cells takes $O(N)$ time [16,17].

The simulation domain is a two-dimensional $L \times L$ box. To avoid spurious boundary effects, the box is periodic on the X and Y directions, meaning that the particles wrap around the box as they move across its boundaries. For the distance calculations between particles, including force calculations, we follow the minimum-image convention, in which a particle interacts with another via its real position or its image depending on which is the shortest [16].

2.2. Particle interaction without excluded volume

Microscopic particles move in an overdamped regime, with no inertia. When subject to a force $\vec{F}$, the equation of motion is simplified to $d\vec{r}/dt = \gamma\vec{F}$, where γ is the mobility. Absorbing the mobility coefficient into the force, which will then have velocity units, in a time step Δt, the integration rule for updating the position $\vec{r}$ of a particle from instant t to $t + \Delta t$ is performed using the Euler–Maruyama method:

$$\vec{r}_i(t + \Delta t) = \vec{r}_i(t) + \vec{F}_i(t)\Delta t + \vec{\xi}\sqrt{D\Delta t}, \tag{1}$$

where $\vec{F}_i$ is the deterministic velocity obtained from the interactions between the particle i and P_i, D is the diffusion coefficient, and $\vec{\xi}$ is a random vector, where the components follow a normal distribution of zero mean and unit variance, and corresponds to a noise added that takes into account the diffusive Brownian motion [18].

The force model we use for the simulations describes the interaction of self-diffusiophoretic active particles [12]. In this model, particles can be of different type, characterized by two charges, α and μ; the former is responsible of creating the concentration field, while the second describes the response of a particle to the field, leading to the following interaction law:

$$\vec{F}_i = \sum_{k \neq i} \mu_i \alpha_k \vec{f}(\vec{r}_i - \vec{r}_k), \tag{2}$$

where $\vec{f}(\vec{r}) = \vec{r}/r^3$ for the studied long range force, while $\vec{f}(\vec{r}) = \vec{r}/r^7$ for the short range interaction. Both long range and short range forces can coexist in the same simulation. Note that if $\alpha_i \neq \mu_i$, the action–reaction symmetry is broken and self-motion is possible. Charged colloidal particles are included in this model if $\alpha_i = \mu_i = q_i$, equal to the electric charge of the particles.

Since the short range force decays much faster with distance compared to the long range force, its calculation considers a cutoff radius from which the value of the force is considered zero, as shown in Fig. 1. The short range force is then computed as:

$$\vec{F}_{ij}(\vec{r}_{ij}) = \begin{cases} \mu_i \alpha_k \vec{f}(\vec{r}_{ij}), & \text{if } r_{ij} \leq r_{\text{cutoff}} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

We used $r_{\text{cutoff}} = 2.5\sigma$ for the simulated short range force in our experiments, where σ is the particle diameter.

2.3. Excluded volume

The simulated particles are represented as hard disks with a uniform diameter σ. Although here we consider only monodisperse colloids, it is direct to extend the method to polydisperse

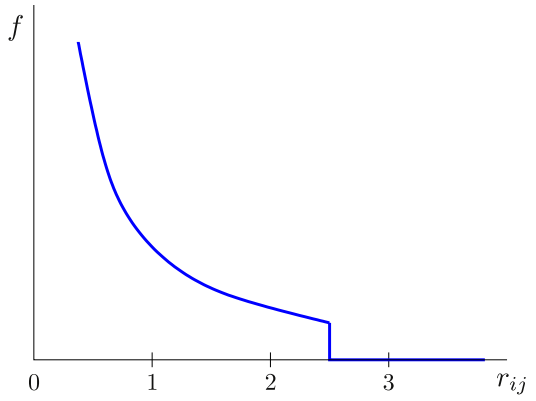


Fig. 1. Cutoff of the short range forces. For distances larger than r_{cutoff}, the force is small and therefore is set to zero to speed up calculations. The jump at r_{cutoff} has been exaggerated for illustration purposes.

systems where radii dot not differ too much. Since the integration rule (1) ignores the excluded volume condition, it can happen that the updated positions produce overlaps between two or more particles, resulting in a physical impossibility. In order to ensure this property, at the end of each time step, the members of all overlapping pairs (p_i, p_j) are moved apart from each other in a way that corrects the overlaps:

$$\vec{r}_1\prime = \vec{r}_1 - \delta^* \frac{(\vec{r}_2 - \vec{r}_1)}{|\vec{r}_{12}|} \qquad \vec{r}_2\prime = \vec{r}_2 - \delta^* \frac{(\vec{r}_1 - \vec{r}_2)}{|\vec{r}_{12}|}, \tag{4}$$

where $\vec{r}_1$ and $\vec{r}_2$ are the original positions and $\vec{r}_1\prime, \vec{r}_2\prime$ the updated positions. If $\delta^* = (\sigma - |\vec{r_{12}}|)/2$, the particles would move in opposite directions from each other along $\hat{n} = (\vec{r}_2 - \vec{r}_1)/|\vec{r}_{12}|$, leaving the particles in tangential contact. If $\delta^* = \sigma - |\vec{r}_{12}|$, the movement is proportional to the magnitude of the previously existing overlap, simulating a bounce effect resulting from the collision at some instant $t^* \leq t + \Delta t$. This last value is the one used for processing the overlaps in the simulation and guarantees that no accumulation is produced at the contact distance.

2.4. Stochastic displacements

The particle displacements on (1) have a random noise component $\vec{\xi}$, modeled as a random variable with standard normal (or Gaussian) distribution with zero mean $\hat{\mu}$ and standard deviation $\hat{\sigma} = \sqrt{D\Delta t}$ (note that these $\hat{\sigma}, \hat{\mu}$ have different meanings from the hat-less σ, μ used on other sections of this paper). Reducing Δt, the deterministic and stochastic displacements in each time step are also reduced. However, for a Gaussian distribution, it is always possible that large values are generated (at the tail of the distribution), leading to excessively large displacements (see Fig. 2). To avoid these problems, the simulation ignores values larger than $3\hat{\sigma}$. This value is derived from the probability of observation X under this model is $P(\hat{\mu} - 3\hat{\sigma} \leq X \leq \hat{\mu} + 3\hat{\sigma}) \approx 0.997$, which means nearly all observed values lie within three standard deviations of the mean. We considered two methods in order to achieve this, as shown in Fig. 3:

(a) Reroll the values outside the range $[-3\hat{\sigma}, 3\hat{\sigma}]$.
(b) Truncate the values to the range $[-3\hat{\sigma}, 3\hat{\sigma}]$.

Both methods produce probability distributions different from each other and from the original; while the first alternative raises the probability of all values in range, the second one raises the probability at the edges. These modifications do not generate a noticeable statistical distortion, since the considered range includes 99.7% of the possible values. In our simulations, we opted for the second method, which turns out to be faster and better suited for parallel execution, since it needs to generate a single random number instead of a variable quantity of random values in the first method.

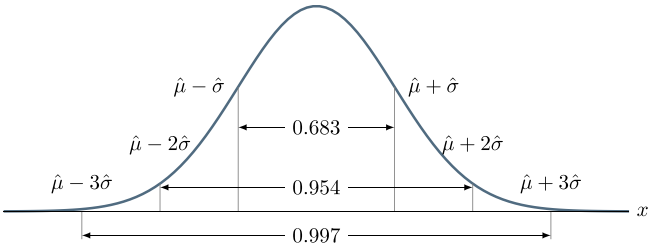


Fig. 2. Normal distribution with mean $\hat{\mu}$ and standard deviation $\hat{\sigma}$. The probability to get numbers in the ranges $[\hat{\mu} - \hat{\sigma}, \hat{\mu} + \hat{\sigma}]$, $[\hat{\mu} - 2\hat{\sigma}, \hat{\mu} + 2\hat{\sigma}]$, and $[\hat{\mu} - 3\hat{\sigma}, \hat{\mu} + 3\hat{\sigma}]$ are 0.683, 0.954, and 0.997, respectively.

3. Related work

For short-range forces calculation, the standard technique is the use of Verlet lists [16,17,19]. The authors in [20,21] parallelize the list construction by having a $O(N^2)$ list of all possible pairs of bodies. A predicate checking closeness between the pair members is evaluated over all elements of the list, which can then be used for a key–value sort to group all the neighboring pairs consecutively in the array. A parallel scan operation allows to get the number of elements that must be copied to the neighbor list. The authors then combine this algorithm with fixed cell partitioning in order to replace distance calculations with less-expensive cell neighborhood checks.

For the parallel n-body simulation, with full calculation of the $O(N^2)$ forces, Nyland et al. [22] developed a grid-style tiling algorithm, reading the particles from the global space and storing them on GPU shared memory, increasing performance as multiple threads read from that space at a lower latency. Partitioning the load/store process on groups of p particles allows fitting an arbitrary input size on the hardware-limited shared memory size. Burtscher and Pingali [23] parallelized the Barnes–Hut simulation [14], which computes an approximation for the force, representing the cell hierarchy kd-tree as multiple arrays for each node field. It uses atomic lock operations to build the tree in parallel, throttling the threads that failed to get the lock so they do not waste bandwidth with unsuccessful lock requests. The tree is then filled with the center of mass data, starting from lower nodes in the tree according to the order of allocation for the scan. Bedorf et al. [11] uses a Z-order curve to sort the particles spatially. Each thread is assigned to a particle, applying a mask value to it to determine the octree cell the particle should be assigned. The linking of the tree is made by assigning a thread to each cell node and then doing a binary search over the corresponding Z-order key to find both the parent and child nodes, if appropriate.

To detect and process collisions, Hawick and Playne [24] developed a multi-GPU algorithm with a tiling scheme similar to the one used by Nyland et al. [22]. If a pair of particles overlap, the associated threads store the index of its colliding neighbor and the time at which the collision occurred. The collisions then are resolved iteratively starting from the earliest, redoing the previous process in order to find possible new collisions.

Finally, for overlap correction, Strating [25] describes a brute-force sequential algorithm that checks all pairs of bodies for possible overlaps and corrects them following Eq. (4). The algorithm may need to iterate an unbounded number of times at each time step because some corrections may generate new overlaps with neighboring particles.

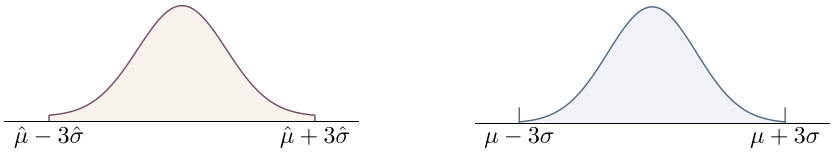


Fig. 3. Distributions that result after discarding values larger than $3\hat{\sigma}$ from the original Gaussian distribution. Two methods are used. Left: Rerolling values out of range. Right: Truncating the generated values. In this case, Dirac-delta contributions of small amplitude appear at $\hat{\mu} \pm 3\hat{\sigma}$.

4. Algorithm

This section describes the parallel algorithm in detail. It includes the generation of the initial data, parallel data structures, overlap detection and correction, and Delaunay condition updates, including how the above processes are mapped into parallel CUDA kernels.

4.1. Overview

The simulation consists of two phases: (i) sequential initialization of the simulation data, followed by a host to device transfer and (ii) a parallel simulation phase, described in Algorithm 1. The initial positions are initialized over a triangular mesh with $N^* \geq N$ vertex, where each vertex represents a particle and their types are assigned randomly according to the specified concentrations. A sample of N particles is selected from the mesh by Reservoir Sampling [26], resulting in the input particle set, which is homogeneous in space.

The n-body algorithm for the long range force is based on a grid-style tiling, which uses the shared memory of the multiprocessor assigned to each thread block to store the particles in groups. In this algorithm each thread is mapped to exactly one particle in the system, and since it is possible to lack action–reaction symmetry on the force, no redundant computation is done unlike the cases where $\vec{f}_{ij} = -\vec{f}_{ji}$.

Once the forces are calculated, the particles are advanced one time step using Eq. (1). As a result, particle overlaps can appear. When Δt is small enough, for hard disks of similar or equal radii, only neighbor particles can overlap. Then, to detect and correct overlaps, instead of a brute-force algorithm that would check all $O(N^2)$ pairs, only neighbors are checked. The Delaunay triangulation [27] is particularly well suited to detect neighbors for monodisperse or slightly polydisperse disks. In dense systems, the overlap corrections can be highly non-local, as the correction of one pair generate a sequence of other overlaps that need correction. Therefore, the computational cost of this stage is not clear a priori, which is the reason why we consider both short and long range interaction forces.

The triangulation is used as input for overlap correction, with the edges connecting spatially close particles for distance checking. The overlaps are solved in pairs of particles (two overlapping particles are considered a single pair), and each particle can participate in only one overlap correction. This raises the need of iterating the previous procedure until there are no overlaps, which is not known to converge. However, the experiments in Section 6, which represent extreme testing conditions, show empirical convergence. Furthermore, similar experiences are reported in [25].

4.2. Data structures

We store the simulation data as a Structure of Arrays (SoA) on global device memory, using total $O(N)$ space. The particle data consists of their position (x_i, y_i) and their charges (α_i, μ_i), stored as floating point vector types.[1] in order to increase bandwidth utilization [28]. We use an additional buffer array for positions so that writes are not done at the same addresses for reads, avoiding a synchronization step. We store the simulation parameters that remain unchanged during a same instance on a constant device memory structure [28], such as $N, D, \Delta t, \sigma$ and derived constants σ^2 and $\sqrt{D\Delta t}$. Additionally, we store the triangulation data using the same scheme as [29].

[1] `float2` or `double2`.

Algorithm 1 Particle system simulation

Input: $P = \{p_1, ..., p_N\}$ list of particle positions
Output: $P = \{p_1, ..., p_N\}$ list of positions updated to current time

```
1:  procedure RunSimulation(P)
2:      Generate starting position of N particles
3:      Build the Delaunay triangulation
4:      for t ← 0 to T_f do
5:          Integrate the N particles on t + Δt          ▷ Algorithm 6
            repeat
                Correct inverted triangles               ▷ Algorithm 4
                Repair triangulation                     ▷ Algorithm 3
            until inverted triangle count is 0
            repeat
                Flip non-Delaunay edges                  ▷ Algorithm 2
                Repair triangulation                     ▷ Algorithm 3
            until non-Delaunay edge count is 0
6:          repeat
7:              Correct overlaps between particles       ▷ Algorithm 5
8:          until overlap count is 0
9:      end for
10: end procedure
```

4.3. Delaunay triangulation

The Delaunay triangulation can be built either constructively or from an existing triangulation. Lawson's algorithm [30] accepts a triangulation as input and transforms it into a Delaunay triangulation via a finite sequence of edge-flip operations [27]. Based on the Lawson algorithm, Navarro et al. [29] developed a parallel implementation for generating quasi-Delaunay triangulations, so it is possible to keep the triangulation updated without need of further host-device memory transfers. These are quasi-Delaunay because exact predicates are too expensive on the GPU; nevertheless, this approximate construction is sufficient for our problem. The initial triangulation is built from scratch on host memory once [31], and is updated on device memory using the Navarro et al. algorithm during subsequent time steps. As the particle system is simulated under periodic boundary conditions, the underlying triangulation must also be periodic [32], which means there are no boundary edges and the number of edges remains invariant.

The triangulation data structure consists of vertices V, edges E and triangles T. An edge $e = (v_1, v_2)$ contains the indices to its endpoint vertices on array V; likewise, a triangle $t = (v_1, v_2, v_3)$ contains the indices of the vertices that compose it. An edge also knows the two triangles adjacent to it: $t_a = (t_{a1}, t_{a2})$ is one of the triangles adjacent to an edge $e_i \in E$. Its components t_{a1}, t_{a2} are indices of the triangles array T. Likewise, $t_b = (t_{b1}, t_{b2})$ represents

Algorithm 2 Flip non-Delaunay edges

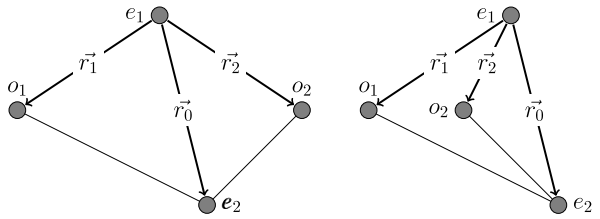
Input: Triangulation $\tau = (V, E)$, array of triangles T, empty array of triangle rotations R, array of triangle locks L
Output: Delaunay triangulation τ, R with recorded triangle rotations

```
1:  procedure DelaunayEdgeFlip(τ, T, R, L)
2:      for all edges e_i in the triangulation in parallel (i ← 0 to |E|) do
3:          Thread i maps to edge e_i
4:          t_a and t_b are adjacent triangles sharing e_i
5:          if e_i is not a Delaunay edge then
6:              Try to get locks L[t_a] and L[t_b]
7:              if thread has locks L[t_a] and L[t_b] then
8:                  Flip e_i
                    R[t_a/3] stores the rotation of t_a with t_b
                    R[t_b/3] stores the rotation of t_b with t_a
9:              end if
10:         end if
11:         Synchronize threads
12:     end for
13: end procedure
```

Algorithm 3 Repair Delaunay triangulation

Input: Triangle array T, Array of triangle rotations R
Output: R reflecting the state of Delaunay triangulation τ

```
1:  procedure DelaunayRepair(R)
2:      for all edges e_i in the triangulation in parallel (i ← 0 to |E|) do
            Thread i maps to edge e_i
            t_a1 and t_b1 are adjacent triangles sharing e_i
            t_ra ← R[t_a1/3], t_rb ← R[t_b1/3]
            if t_a1 ≠ t_ra or t_b1 ≠ t_rb then
                Repair t_a2 and t_b2
            end if
3:      end for
4:      Synchronize threads
5:  end procedure
```



(a) Starting situation. (b) Inverted triangle.

Fig. 4. Inverted triangle detection using barycentric coordinates. Particle sizes are scaled down compared to distances.

the other triangle neighboring the same edge, and is empty if the edge belongs to the triangulation boundary for non-periodic triangulations. When the triangulation is valid, the endpoints of an edge may be accessed directly by v_1 and $v2$, or indirectly by either of the pairs $(T[t_{a1}], T[t_{a2}])$ or $(T[t_{b1}], T[t_{b2}])$. However, this is not the case after an edge flip changes the member vertices of t_a and t_b, meaning the triangulation is in an invalid state as the triangles have outdated member information. An array R of size equal to the number of triangles records the triangle rotations associated with each edge flip, which then are used to restore the triangles whose vertices are no longer consistent with the updated state of the triangulation after the flip. Note that for a given $t_i \in T$, the index of its triangle in R can be found as $i = t_i/3$, as a triangle is composed of three consecutive indices in T. A more detailed description of the involved processes and data structures is contained in [29].

If an edge violates the empty circle property after the integration step, it needs to be flipped because it is not a Delaunay edge, which renders it invalid for overlap correction. The data structure allows a single thread to get the two triangles sharing an edge tested for the Delaunay condition, which allows the parallel algorithm to use a mapping of one thread per edge in the triangulation. It is not possible to perform a flip in parallel of two edges sharing a same triangle without compromising the consistency of other neighboring edges, thus creating a race condition. This is prevented by implementing an array of locks, in where each element represents a triangle in the structure and each edge needs to hold the locks corresponding to its two neighboring triangles before proceeding with the flip operation. If a thread is not capable of claiming both locks, it will skip work and the function will iterate until there are no more edges to be flipped.

After each edge flip iteration, the edges neighboring the flipped edge end up with outdated information, still containing references to triangles formed by the unflipped, obsolete edges. The triangle array T and the references to the neighboring triangles for each edge have redundant information contained in the triangle pairs (t_{a1}, t_{b1}) and (t_{a2}, t_{b2}), which allows the algorithm to check for these inconsistencies efficiently. Using this, a second kernel mapping one thread per edge updates the triangle references of the edges that were not flipped. These edges may have changed their neighbor triangles, because of the new triangles generated by the flip edges [29].

4.4. Inverted triangle detection

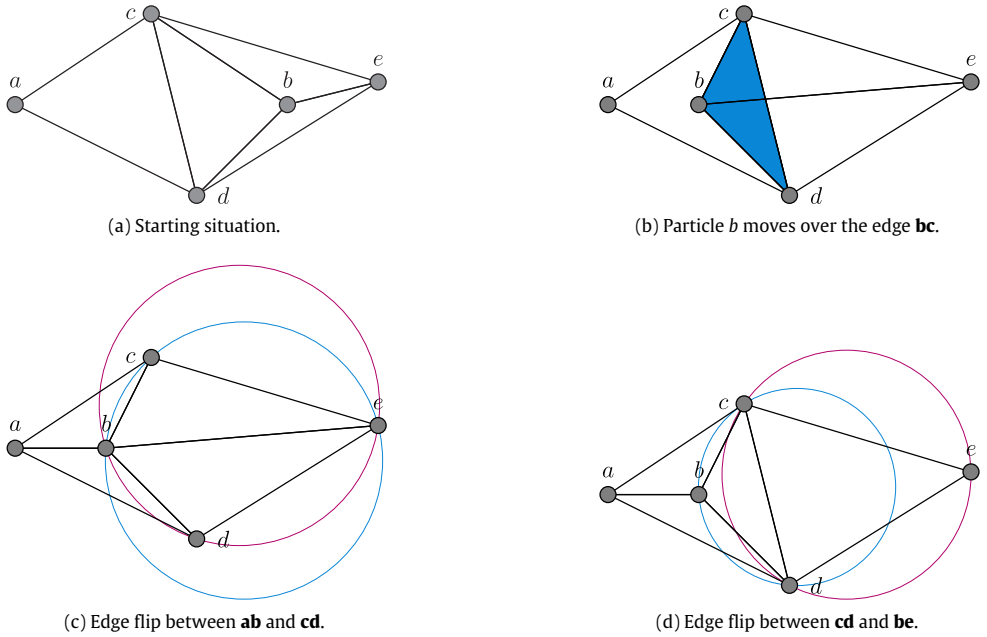
As the particles move due to integration, the vertices of the triangulation may end in an invalid state if one or more triangles end up inverted. The stored data structure would stop being a triangulation, and as such it cannot be passed as input to the Navarro et al. algorithm. There are two ways an inverted triangle can be formed:

1. A pair of particles ends up in opposite directions after integration, meaning they went right through each other.
2. A particle ends up at the opposite half-plane of an edge in the triangulation, which can also be interpreted as the particle crossing or going through that edge.

The first problem is a physical impossibility that violates the excluded volume property, and as such it must not be allowed. The criterion $\vec{r}_{ij}^{\,0} \cdot \vec{r}_{ij}^{\,1} < 0$ allows to check if any such particles existed during the current time step, where $\vec{r}_{ij} = \vec{r}_j - \vec{r}_i$ is the distance vector between the particles, $\vec{r}_{ij}^{\,0}$, $\vec{r}_{ij}^{\,1}$ are the distances before and after integration respectively. The test is ran as a CUDA kernel using a mapping of one thread per triangulation edge, meaning each thread compares a pair of nearby particles. Once an invalid movement is detected, the last positions are discarded and integration is repeated with a lower Δt value than the currently used.

The second problem requires different treatment whether the particle has crossed a single edge or more of them. The first case is the most common and is equivalent to a point-in-triangle check, where a positive result means a neighboring particle crossed over one of the edges composing the triangle. Fig. 4 shows cases where the check is both true and false (left and right figures, respectively).

The data structure for edges holds the indices of its endpoints and the two opposite vertices; for example, in Fig. 4 the edge $e = (e_1, e_2)$ also holds the vertices (o_1, o_2) opposite to it, which is all the data needed for executing the test. Then, the point-in-triangle test can be executed in parallel on a per-edge basis, where



(a) Starting situation.

(b) Particle b moves over the edge **bc**.

(c) Edge flip between **ab** and **cd**.

(d) Edge flip between **cd** and **be**.

Fig. 5. Inverted triangle correction. The cyan shaded triangle was inverted by the movement of particle b. Particle sizes are scaled down compared to distances.

Algorithm 4 Correct inverted triangles

Input: Invalid triangulation $\tau = (V, E)$, array of triangles T, array of triangle rotations R, array of vertex locks L

Output: Delaunay triangulation τ, R with recorded triangle rotations

```
1:  procedure DelaunayTriangleCorrect(τ, T, R, L)
2:      for all edges e_i in the triangulation in parallel (i ← 0 to |E|) do
3:          Thread i maps to edge e_i
4:          t_a and t_b are adjacent triangles sharing e_i
5:          if e_i fails the inverted triangle test then
6:              Try to get locks L[t_a] and L[t_b]
7:              if thread has locks L[t_a] and L[t_b] then
8:                  Flip e_i
                    R[t_a/3] stores the rotation of t_a with t_b
                    R[t_b/3] stores the rotation of t_b with t_a
9:              end if
10:         end if
11:         Synchronize threads
12:     end for
13: end procedure
```

each thread is assigned to a single edge in the triangulation. This matches the mapping used in [29], meaning the inverted triangle correction algorithm (Algorithm 4) is identical to the edge flipping algorithm (Algorithm 2), save for the predicate used to determine which edges have to be flipped.

As the triangle correction procedure involves edge flips, the triangle data needs to be updated by running Algorithm 3. Finally, the test for inverted triangle correction is implemented efficiently using barycentric coordinates d, s, and t on triangles (see Fig. 4). The data structure for the edges does not ensure that the triangles used during the test will be built in a consistent ordering (clockwise or counter-clockwise), so the signs of the cross products between distance vectors enable performing the check regardless of the actual ordering of the vectors. This is shown at the `flip` predicate, which takes the four vertices associated to an edge to answer if an edge needs to be flipped. The lines of the final logical operation check if o_1 lies inside triangle (e_1, e_2, o_2), or if o_2 lies inside (e_1, e_2, o_1). Also, for each operation, the predicate also considers the two possible orderings of the vertices of the respective triangle, which accounts for the four lines of logical operations in the predicate.

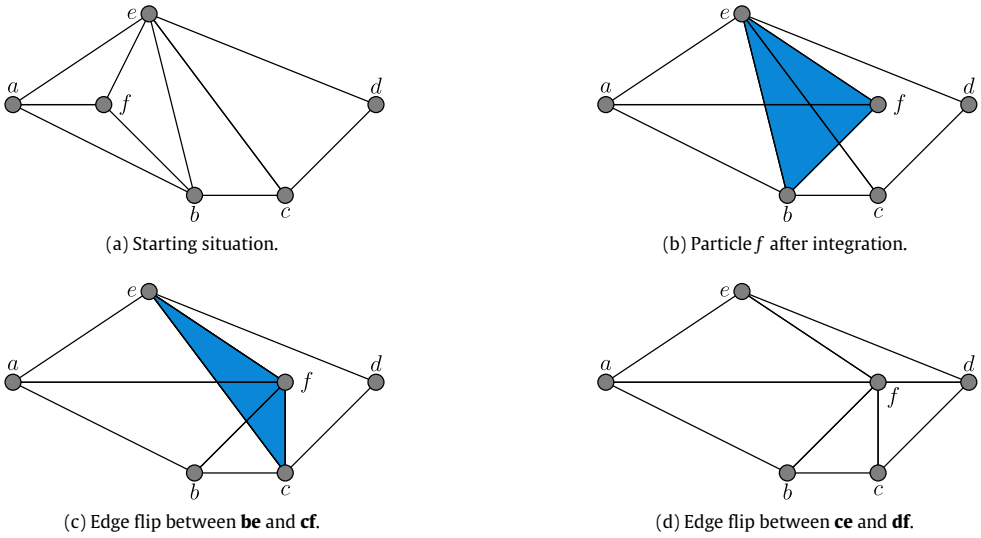
$$d = \vec{r}_2 \times \vec{r}_0 \qquad s = \vec{r}_1 \times \vec{r}_0 \qquad t = \vec{r}_2 \times \vec{r}_1 \tag{5}$$

$$\text{flip}(\mathbf{e_1}, \mathbf{e_2}, \mathbf{o_1}, \mathbf{o_2}) = \begin{cases} (d < 0 \wedge s \leq 0 \wedge t \leq 0 \wedge s + t \geq d) \vee \\ (d > 0 \wedge s \geq 0 \wedge t \geq 0 \wedge s + t \leq d) \vee \\ (s < 0 \wedge d \leq 0 \wedge -t \leq 0 \wedge d - t \geq s) \vee \\ (s > 0 \wedge d \geq 0 \wedge -t \geq 0 \wedge d - t \leq s). \end{cases} \tag{6}$$

It is worth noting that the movement of point b across edge (c, d) creates an intersection between it and edge (b, e), as shown in Fig. 5. The edge flip between (a, b) and (c, d) removes the inverted triangle, restoring the local triangulation. The triangulation may still not satisfy the Delaunay property, so additional edge flips may be needed on further steps. For this stage, we use the Navarro et al. algorithm [29].

The point-in-triangle equivalence holds true only when a particle crosses no more than one edge. If not, it is possible to repair the triangulation via a sequence of edge-flips, where the number of flips is number of edges crossed by the particle. Fig. 6 shows a situation where particle f moves across edges (b, e) and (c, e), needing two consecutive flips in order to restore the local triangulation. Upon visual inspection, it is clear that the offending edges need to be flipped because they intersect with edge (a, f), a situation that invalidates the triangulation.

In order to execute this sequence of flips, an algorithm would need to identify the sequence of edges that were trespassed by an offending particle. The currently implemented triangulation data structure has not enough locality information to create this set efficiently, so function (6) cannot properly evaluate this case or similar movements across further distances, even when it should be possible to do so. The simplest possible test would be checking if the two opposite vertices to an edge lie at the same side after integration, but this is not enough to identify the single edge that needs to be flipped. For Fig. 6b), note that all three edges of the inverted triangle (b, e, f) satisfy this condition: while c and f are on the same side of (b, e) (the right answer), so are a and b regarding (e, f), and a and e regarding (b, f).



(a) Starting situation.

(b) Particle f after integration.

(c) Edge flip between **be** and **cf**.

(d) Edge flip between **ce** and **df**.

Fig. 6. Inverted triangle correction with two edge flips. Particle sizes are scaled down compared to distances.

However, when a movement of this kind is detected, the algorithm can undo the last integration and redo it with a lower Δt, since small time steps guarantee that this situation is extremely unlikely to happen. This means the triangulation can be repaired via edge-flipping if the inverted triangles were produced by small perturbations, which is the case if small time steps are used when simulating. Moreover, movements large enough to produce these kind of problems are undesirable for the accuracy of the simulations for the studied problem, which makes the current solution sufficient for small time-step simulations. More details on this issue are included in the discussion section.

4.5. Overlap correction

The overlap correction algorithm (Algorithm 5) uses the topological information contained in the edges of the Delaunay triangulation, which allows for each particle fast access to the neighborhood of particles that may be overlapping with it. The algorithm maps threads to edges in such a way that each thread handles one edge of the triangulation. A thread gets the positions of the particles that form the edge, checking if there exists an overlap between them ($r_{ij} < \sigma$). If the check is positive, the algorithm computes the displacements of the involved particles according to (3). Since the same particle can be part of many edges at once, the algorithm sums atomically the displacements in a global array, in order to avoid concurrency hazards. As stated on the main algorithm, the corrections must be made until there are no more detected overlaps, which is indicated by the `workFlag` global variable that communicates with the host for control flow. If the flag is false, none of the edges has length less than σ, meaning there are no more overlaps present in the system.

The distance vector calculation must follow the minimum image convention imposed by the periodical boundary. The origin is placed at the lower left corner of the square simulation box with size L, which allows to define the distance formula without conditional operators that would cause warp divergence during kernel execution.

Once the algorithm computes the total displacements, it maps each thread with a particle in the same way as described previously for edges. Each thread then updates the position of its particle, applying the periodic boundary conditions when necessary. It is possible that the updated positions may still have some of the previous overlaps or even have some newly generated ones. In this case, the algorithm repeats the previous process until no overlaps are present.

If a particle overlaps with multiple particles on the same general direction and orientation, the net displacement is likely to be more than needed if there are no overlapping particles in the opposite general direction (see Fig. 7, in where particle C is overlapped twice from the same direction by A and B). Those larger, more than needed displacements can generate new overlaps if the neighborhood is dense enough. Eventually, the correction of those new overlaps can result in an instability, where the displacements increase with alternating sign and the iterative procedure does not converge. A solution is truncating the displacement with the heuristic value $\sigma/4$ (half the particle radius), preventing the emergence of the instability.

Finally, it is worth mentioning that the parallel correction algorithm presented here does not correspond to a parallelization of the Strating's sequential algorithm [25], which displaces particles sequentially, while in our case the displacements are added and performed in parallel. Hence, due to the chaotic dynamics of the system, the small differences in these algorithms will produce different outputs for finite Δt.

Algorithm 5 Overlap correction loop

Input: Triangulation $\tau = (V, E)$.
Output: D displacements over each particle

```
1:  procedure correctOverlaps(V, E, D)
2:      for all edges e_i in the triangulation in parallel (i ← 0 to |E|) do
3:          Thread i maps to edge e_i = (v_1, v_2)
4:          v_1, v_2 ∈ 0 to |V| are indices of the endpoints to e_i
5:          r⃗_1 ← V[v_1], r⃗_2 ← V[v_2]
6:          r⃗_12 ← distance(r⃗_1, r⃗_2)
7:          if |r⃗_12| < σ then
8:              workFlag ← true
9:              δ_i ← σ − |r⃗_12|
10:             atomicAdd(D[v1], −δr⃗_12)
11:             atomicAdd(D[v2], δr⃗_12)
12:         end if
13:     end for
14: end procedure
```

(a) Starting situation. (b) After correction.

Fig. 7. Possible instability with the parallel overlap correction. The displacement over C caused by the overlaps with A and B can be greater than needed, which can generate a larger overlap with a neighboring particle D. This problem is prevented by truncating the displacements to a maximum amount.

Algorithm 6 Particle system integration

Input: P_0 particles
Output: P_1 updated particle positions

```
1:  procedure INTEGRATE(P_0, P_1)
2:      for all particles p_i in the system in parallel (i ← 0 to |P|) do
3:          Thread i maps to particle p_i = (x_i, y_i, α_i, μ_i)
4:          l_i is the current lane id for thread i
5:          v_i ← 0                                   ▷ Particle velocity
6:          for j ← 0 to |P|; j ← j + warpSize do
7:              p_j ← P_0[j + l_i]
8:              for k ← 0 to warpSize do
9:                  p_k ← shuffle(p_j, k)
10:                 r_ik ← distance(p_i, p_k)
11:                 v_i ← v_i − r_ik · (α_i μ_k)/r_ik^3
12:             end for
13:             Synchronize threads
14:         end for
15:     end for
16:     P_1[i] ← b_i + v_i Δt + ξ_i √(DΔt)
17: end procedure
```

4.6. Long range forces

An improvement to the long range force calculation (Listing 6) consists on using the intrinsic warp shuffle instruction, which allows a thread access to the registers of other threads belonging to the same warp. Each thread is assigned a lane number l_i that identifies it from the other warp members, allowing them to read different particles from global memory. Then, each warp member takes turns in propagating the data of its corresponding particle to the other threads, who can read it via the shuffle instruction by passing as argument the lane number of the thread currently in turn. Once the whole warp has shared the data among its members, each member reads a particle from global memory and repeats the same process until all particles have been visited. The main advantage of this optimization is a greater efficiency of memory accesses, since most of the time the threads are sharing data at registry level instead of more expensive load requests on global memory. Also, the concurrent execution of the warp members makes unnecessary the explicit synchronization at the inner warp shuffle loop.

The integration and interaction functions calculate Eqs. (1) and (2) respectively, while the random noise value is generated once per thread. It is worth noting from Eq. (2) that μ_j is not used, which allows the algorithm to skip the shuffle instruction for fetching `bj.w`.

5. Implementation

The parallel algorithms described in the previous section were implemented on CUDA 7.5 and C++11, using function templates to choose between `float` and `double` precision formats at compile time. We use the CGAL library [31] to create the 2D periodic Delaunay triangulation, which is then sent to device memory alongside the particle data before starting the simulation. The random numbers used on the parallel implementations when initializing the starting positions and generating noise during integration are created with the XORWOW pseudorandom number generation algorithm of the `cuRAND` library [33], using the host and device APIs respectively. Each configuration has two particle types, although the program can support a variable number of particle types for simulating. For comparison purposes, we also implemented a fully sequential long range forces algorithm with the overlap correction discussed in [25], and a parallel short range forces algorithm using Verlet lists and a discrete grid over the simulation box. The neighbor list computation during the Verlet lists construction in parallel is similar to the one described in [34], grouping together all the particles that belong to the same cells.

When the simulation finishes, the final positions are brought back to host memory and written to an output file. The visualizations in Figs. 8, 13–15 were generated reading the respective output files.

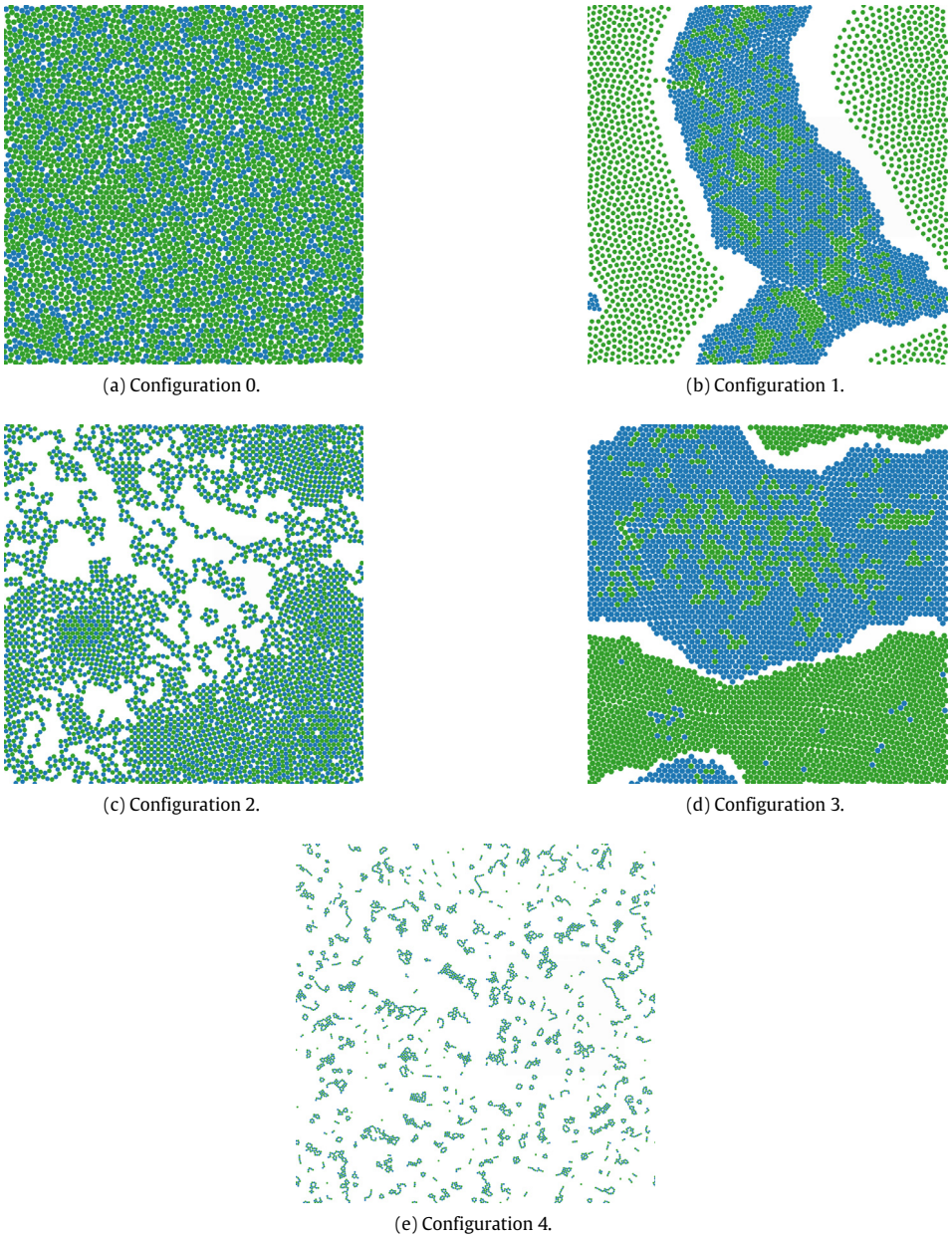
Table 1
Parameters used for the tests, where each configuration is identified by a digit and all of them contain two types of particles. ϕ_i is the concentration of particles of type i, α_i, μ_i are the charges used in the force calculation, and ρ is the area fraction of particles on the simulation box.

config	ϕ_1	α_1	μ_1	ϕ_2	α_2	μ_2	ρ
0	0.7	1	1	0.3	1	−1	0.79
1	0.5	1	1	0.5	1	−1	0.52
2	0.5	1	1	0.5	−1	−2	0.52
3	0.5	−1	1	0.5	1	−2	0.79
4	0.5	1	1	0.5	−1	−4	$8.73 \cdot 10^{-2}$

6. Performance results

Parameters. We generated inputs for 11 different values of N and 5 parameter configurations, as described in Table 1. The starting positions are generated semi-randomly as described in Section 4, keeping the same seed value for the random number generator across all simulation instances.

Each configuration has two types of particles with charges α_i, μ_i. The fraction of particles of each type is given by $\phi_i = N_i/N$, where N_i is the number of particles of each type and $N = N_1 + N_2$ is the total number of particles. The area fraction $\rho = N(\sigma/2)^2/L^2$ is a measure of the particle density. To study scaling times, we change



(a) Configuration 0.

(b) Configuration 1.

(c) Configuration 2.

(d) Configuration 3.

(e) Configuration 4.

Fig. 8. Particle positions after 10^4 iterations, with $N = 4096$ and $\Delta t = 0.01$. Type 1 particles are in green and type 2 in blue. The configurations and particle types for each system are described in Table 1. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the length of the simulation box to keep density constant when increasing N:

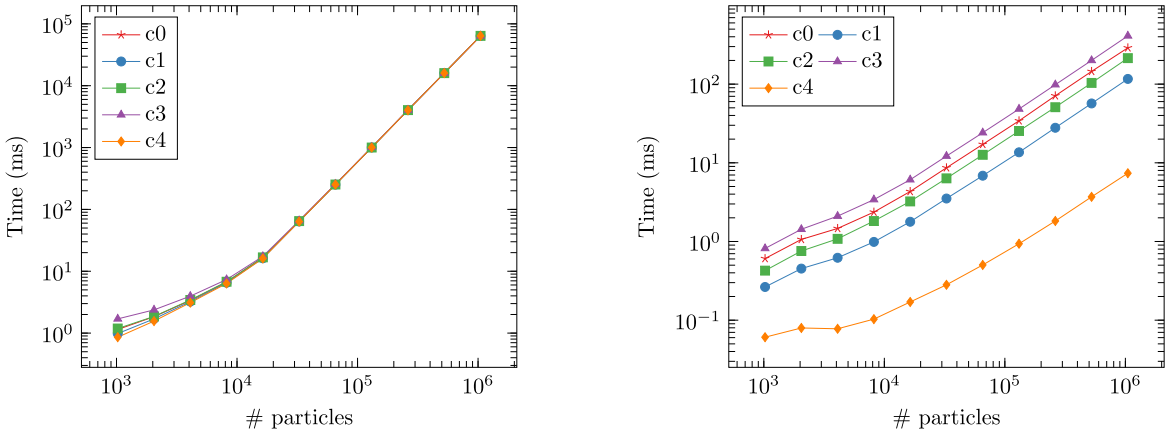
$$L(N) = \sqrt{\frac{N\pi(\sigma/2)^2}{\rho.}} \tag{7}$$

The values for N start at 2^{10}, raising the exponent by 1 until $N = 2^{20}$. Finally, we kept constant the values for $\sigma = 1$, $\Delta t = 0.01$, $D = 0.01$, $\delta = 1.0$ for all configurations and input sizes.

Fig. 8 displays the particle positions after 10^4 time steps for each configuration. The election of the parameters help to test the algorithms under different conditions of fluidity, density and homogeneity. For c0, there is an asymmetric attraction between particles of type 1 and 2, resulting in a homogeneous mixture, with fluid-like motion. In c1, there is a larger concentration of type 2 particles, which self-attract forming a dense cluster, segregated from type 1 particles, which self-repel forming in a gas-like state. In configuration c2, equal particles repel, while dissimilar particles attract, favoring the formation of chain-like structures, where 1 and 2 particles alternate. In c3, the situation is the opposite, where equal particles attract, while dissimilar particles repel, leading to the formation of dense segregated clusters. Finally, the interactions in c4 are analogous to those of c2, in a dilute regime, resulting in the formation of small clusters.

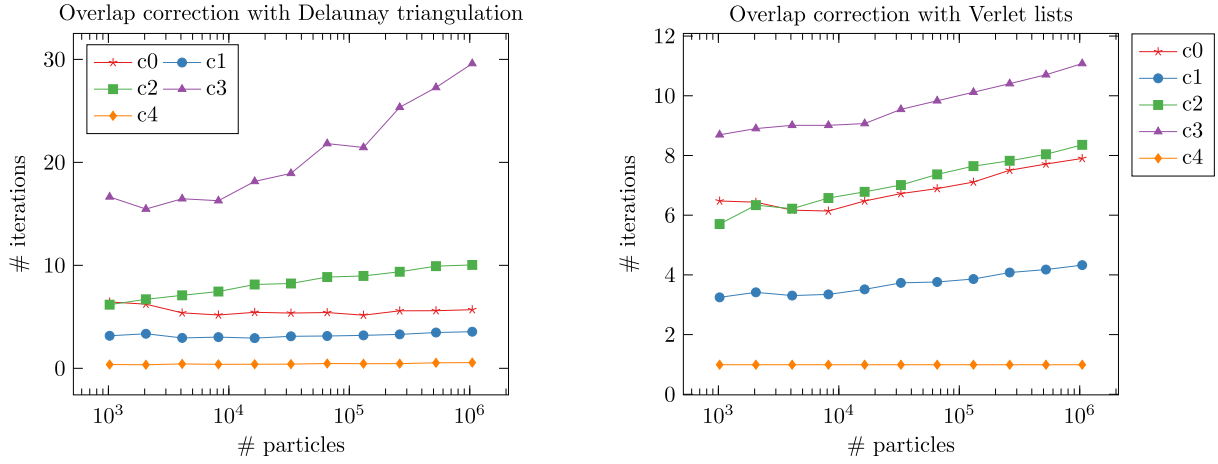
System. We ran the tests on a machine with a Tesla K40c GPU and an Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60 GHz. The tests for both the sequential and parallel implementations were made on the same machine.

Compilation. We compiled the program using `nvcc V7.5.17` with compiler options `--std=c++11 -gencode arch=compute_30,code=sm_30`. For the sequential code, we used `g++ 4.9.2` with options `--std=c++11 -O3`.



(a) Long range forces. (b) Short range forces.

Fig. 9. Average execution time per time step for simulations using long and short range forces, using the parameters shown in Table 1.



(a) Long range forces. (b) Short range forces.

Fig. 10. Average overlap correction iterations per time step for short and long range forces simulation, using the parameters shown in Table 1.

Metric. We ran simulations for 100 iterations, long enough to ensure that particle collisions happen frequently, except for the first iterations where the bodies are separated from each other. There we compute the average execution time and iteration averages for overlap correction cycles and edge flips.

The average execution time per time step, presented in Fig. 9, shows two interesting features. First, for the case with long range forces the execution time is $O(N^2)$, while for short range forces it is $O(N)$. Since both include the overlap correction algorithm, this implies that the execution time for the later is $O(N)$. Second, except for small systems, in the case of long range forces the execution time does not depend on the configuration, while for the short range forces, there is a clear dependence, with increasing complexity for c_4, c_1, c_2, c_0, c_3 (the same order of complexity is observed for long range forces at small N). This result is consistent with a cost $O(N)$ for the overlap correction, with a prefactor that may depend on the density and extension of the clusters.

To study the dependence of different configurations on the complexity of the overlap correction, in Fig. 10 we plot the average per time step of iterations needed to correct all overlaps. The increasing complexity for c_4, c_1, c_2, c_0, c_3 is consistent with the previous results in Fig. 9, because in c_3 most of the particles participate in corrections, while in c_2 almost half of the particles are excluded due to repulsion between same-type particles. However, this does not explain why c_4 has the least complexity factor, even though half of the particles overlap. This happens because all the overlaps on c_4 are corrected on the first iteration, which is probably due to the small size of the clusters. The number of iterations follow the same order in complexity as the execution time. Except for c_4 where clusters are disconnected, the number of iterations grow with N. This effect is due to the percolation of the large clusters, which cover the entire box and, therefore, the corrections become non-local and system size dependent. This growth is nevertheless weak, following an approximate logarithmic law. It is also noteworthy that the curves for long range forces are constant on c_4, c_1 and c_0, slightly grow on c_2 and is relatively greater on c_3.

We also measured the performance of the Delaunay triangulation update algorithm, reporting the average of edge flip iterations made for both inverted triangle corrections and Lawson's algorithm. The curves obtained in Fig. 11 are less regular than the previous results, but keep the same general tendency. Unlike the curves for overlap correction, on where c_4 shows much smaller values than the other configurations, here the curve is comparable to c_0 and c_1. This happens because the underlying triangulation for c_4 has a great number of slivers, formed by the small particle density that forms relatively long edges. Then, according to the inverted triangle condition in Section 4, it is more likely for c_4 to produce inverted triangle than the other configuration, whose

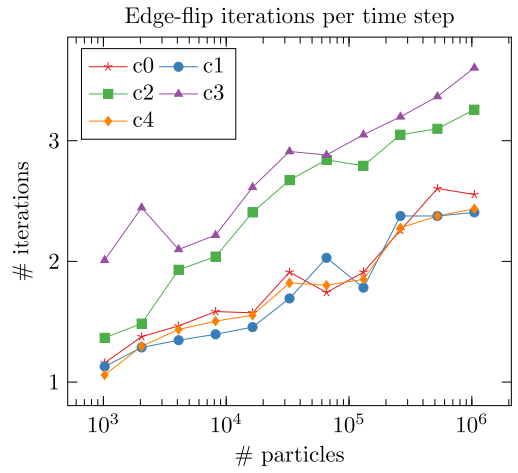


Fig. 11. Average edge-flip iterations per time step for long range forces integration, using the simulation parameters shown in Table 1. The short range forces algorithm is not analyzed, since it does not use Delaunay triangulations.

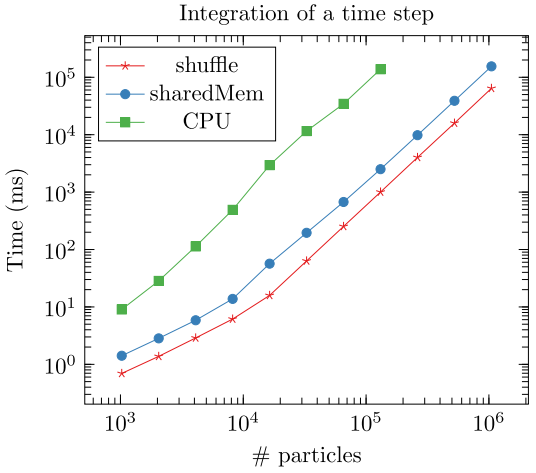


Fig. 12. Comparison between execution times in milliseconds for both implementations of the quadratic n-body algorithm, using configuration c_0 described in Table 1.

triangles are more equilateral. The average for edge-flip iterations for long range forces has linear growth for all configurations, noting that c_2 and c_3 are the hardest cases to solve, as is the case in Fig. 10. Though the number of iterations grows with N, it still remains negligible regarding the total time of a time step, so it is not a priority target for optimization.

Finally, in Fig. 12 we compare the n-body algorithms for long range forces, used on the different implementations without considering overlap corrections. `shuffle` is the presented optimization using warp-shuffle, while `sharedMem` is the GPU device memory algorithm described in Section 4, observing a performance improvement of up to 2.4 times from optimizing the parallel implementation for all tested values of N. It is also noteworthy that the optimized n-body algorithm allows simulation of $N = 10^6$ particles at the same time that the sequential implementation solves the problem for 10^5 bodies. For input sizes relevant to this study ($N \geq 10^4$), the time used by the sequential implementation is two orders of magnitude higher than the parallel solution, which allows the simulation of bigger particle systems for a longer physical time.

7. Validation

In order to verify that the developed overlap correction algorithm is efficient enough, we made two validation experiments. First, we test the locality of the correction, that is, how far it propagates through the system. For configurations $c0$ and $c3$ we print the result after 10^5 time step iterations, painting with red the particles that took part in overlaps during the last simulated time step. Particles that did not take part in overlaps were painted green, so that every particle has a color. We repeat the process for decreasing values of Δt, expecting that the number of overlaps will decrease as the time step produces smaller movements. The results in Fig. 13 allow us to verify the complexity factor associated to each configuration that shows up on the previous performance curves. Configuration $c0$ involves much less particles on overlap corrections than $c3$ upon lowering the time step. This decrease in execution time by reducing Δt does not compensate, however, for the larger number of steps that are needed to achieve a specified physical time.

The second validation consists in testing the overlap correction on another colloidal model. We consider the Active Brownian Particle (ABP) model [4], where particles move in 2D with velocities of fixed magnitude V_0, with a direction that is specified by the director angle θ_i. The integration rule for the positions after an interval Δt is:

$$\vec{r}_i(t + \Delta t) = \vec{r}_i(t) + V_0(\cos\theta_i\hat{x} + \sin\theta_i\hat{y})\Delta t. \tag{8}$$

In the same time interval, the angles θ_i are subjected to diffusive rotational Brownian motion, of amplitude D, and therefore evolve as:

$$\theta_i(t + \Delta t) = \theta_i(t) + \sqrt{2D\Delta t}\, n_i, \tag{9}$$

where n_i is a random Gaussian variable of zero mean and unit variance.

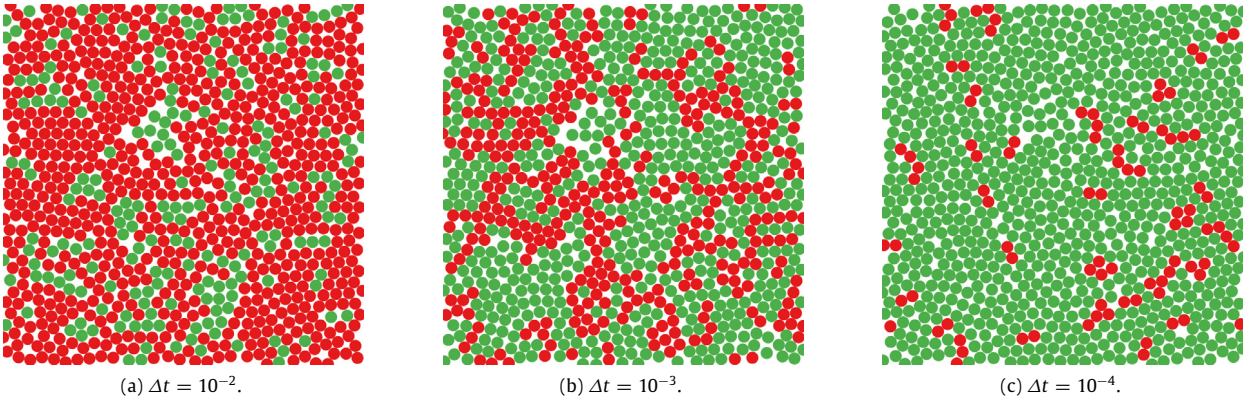
We simulate the system with the same parameters used in Ref. [35], for two different packing fractions, obtaining the same phenomenology. At large packing fractions, the system evolves to the formation of a dense percolating dynamic cluster (see Fig. 14). Reducing the packing fraction, small clusters form, which merge in a slow coarsening process in the course of time as shown in Fig. 15.

8. Discussion

We presented algorithms for simulating colloidal particles subject to Brownian motion, interacting with short or long range force interactions, and presenting excluded volume. The overlap correction algorithm using periodic Delaunay triangulations is a novel method. The algorithms implemented in CUDA for simulation are fully parallel, transferring data back to host only for measurements or outputs. The overlap correction algorithm can be used independently from the forces calculation, allowing to simulate different colloidal models including charged particles or self-propelled active systems. The Delaunay triangulation and the parallel edge-flip algorithm proved to be useful for solving overlaps efficiently. This opens the possibility for using the Delaunay triangulation for solving related problems in the simulation, such as short range force calculation or approximated n-body simulations. The parallel n-body implementation was also successfully adapted and optimized to the particular conditions of colloidal particles, which opens up simulations of up to two orders of magnitude the number of particles used in the previous sequential implementation.

8.1. Considerations for non-periodical triangulations

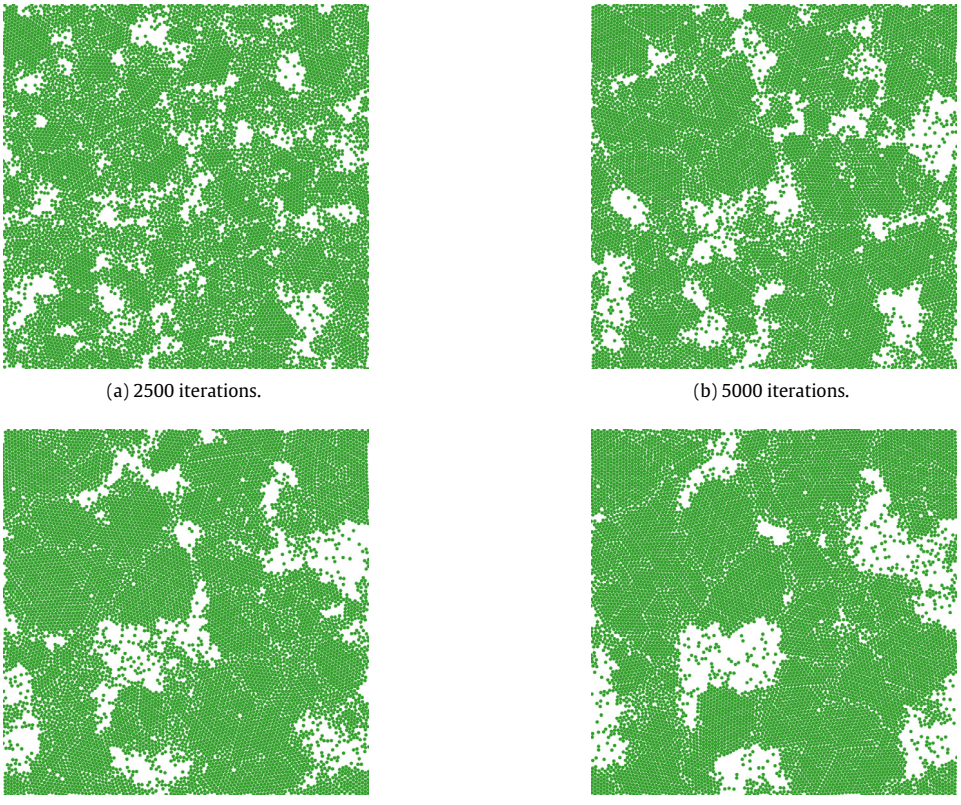
While the problem presented on this work considers periodic boxes, under some conditions it is desirable to model non-periodic systems, hence needing Delaunay triangulations without periodic boundaries. An advantage of periodic triangulations comes from the fact that each edge $e \in E$ is shared by two triangle faces



(a) $\Delta t = 10^{-2}$. (b) $\Delta t = 10^{-3}$. (c) $\Delta t = 10^{-4}$.

(d) $\Delta t = 10^{-2}$. (e) $\Delta t = 10^{-3}$. (f) $\Delta t = 10^{-4}$.

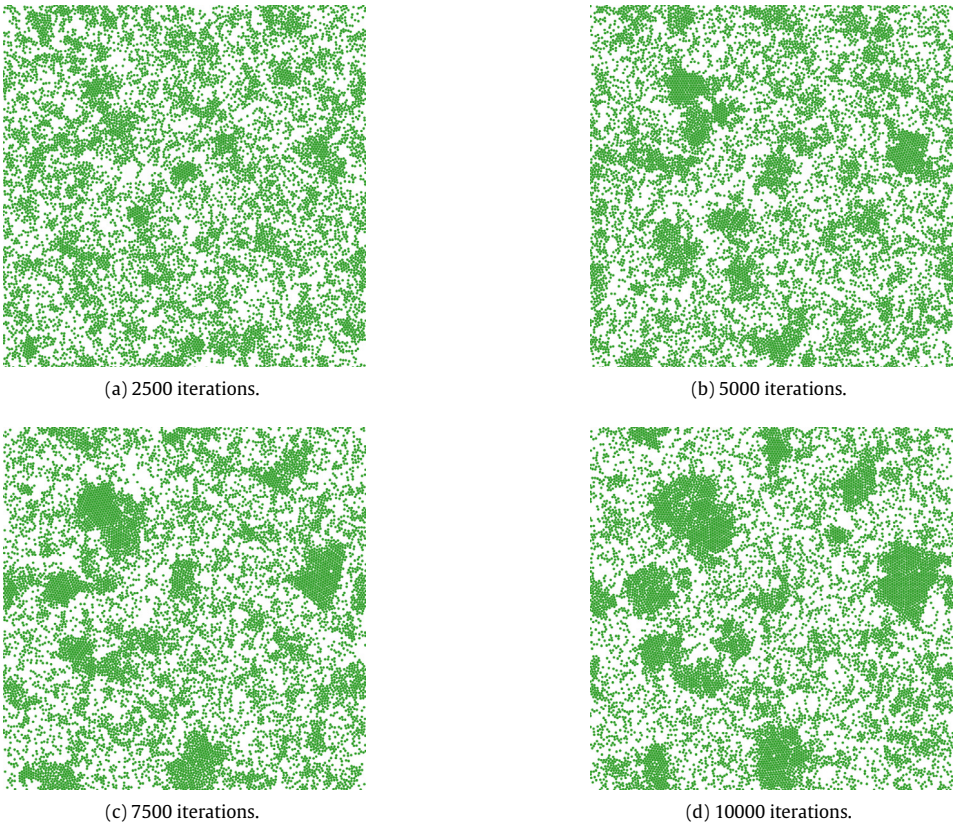
Fig. 13. Overlap correction locality visualizations of configurations $c0$ and $c3$. The red particles participated in at least one overlap correction on the same time step, while green particles did not. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



(a) 2500 iterations. (b) 5000 iterations.

(c) 7500 iterations. (d) 10000 iterations.

Fig. 14. Snapshots of an ABP system $N = 10^4$, $L = 105.9$, $D = 0.01$ and packing fraction $\rho = 0.7$.



(a) 2500 iterations.

(b) 5000 iterations.

(c) 7500 iterations.

(d) 10000 iterations.

Fig. 15. Snapshots of an ABP system rescaled to fit the packing fraction $\rho = 0.4$. The other parameters are the same as presented in Fig. 14.

$f \in F$, which can be expressed with the ratio $|E| = 3|F|/2$. As the edge flips preserve the number of triangles, the number of edges also remains the same, which does not hold true for boundary edges in non-periodical triangulations (consider the case where an internal vertex moves right onto a boundary edge). In this case, the triangulation cannot be maintained using only edge flips.

This problem is solved by adding extra triangulation vertices not representing any particle in the system, placing them outside the corners of the simulation box. The new convex hull becomes a square with a bigger size than the simulation box, forming an impassable barrier as the particles cannot physically trespass the non-periodic simulation boundaries. This allows to preserve the edge-triangle ratio in the triangulation, which in turn allows the edge flip algorithm to correctly update the triangulation at each time step. The new vertices can be simply excluded from physical operations such as integration and overlap correction, as they do not represent colloidal particles.

8.2. Computational considerations for a multi-GPU implementation

The extension to a multi-GPU implementation requires several technical considerations that are not straightforward, therefore some insights are given to parallelize both particles and triangulation. First of all, the parallel design has to consider a distributed memory scenario where parts of the problem are assigned to different GPUs, i.e., the particle set with its corresponding triangulation is not global anymore. In this case it is the data domain the one that can be partitioned, since the functional domain has time-dependencies coming from the physical simulation process. An efficient partition scheme of the data must satisfy the following three properties if possible; (1) work balance, (2) locality and (3) minimal communication.

A reasonable approach for balanced work is to subdivide the set of particles into subsets of similar size and assign one for each GPU. Locality adds another degree of difficulty as the partition must be made in such a way that the subsets of particles belong to continuous two-dimensional regions of the whole domain, from which Delaunay sub-triangulations can be built. Such partition may be achieved with the help of a quad-tree or kd-tree. All integration, corrections and Delaunay updates that are local to each GPU would work with a high level of efficiency. Lastly, the inter-GPU communication coming from the interactions and updates between particles of different sub-sets and sub-triangulations can be kept low by using a hierarchical multi-GPU design based on the organization of nodes and GPUs per node. Lastly, the possibility of asynchronous computation among GPUs may provide additional performance benefits as long as the physical model allows it.

8.3. Considerations for the inverted triangle correction

As shown in Section 5, it is possible to correct inverted triangles by executing a specific sequence of edge flips. However, finding such method in the context of parallel computing has proven to be a non-trivial task and an interesting topic for further research. Finding the inverted triangles is easily done but not sufficient, as a possible implementation would also need to find the specific edge among the three that compose each respective triangle. From the example in Fig. 6, it is possible to notice that the flipped edges come from the same direction, which hints at a topological nature of the problem. Then, an efficient solution would likely need additional topological information in the triangulation data structure, such as the list of neighbors to each vertex, or the list of vertices that have it as endpoint.

Finally, it is worth noting that the intermediate flips in the sequence all produce inverted triangles, which can be corrected by

successive executions of the inverted triangle correction algorithm until all crossed edges are flipped. This means the current implementation has a natural mechanism that the proposed method could use to handle multi-edge crossings, that while inefficient (a single kernel of thousands of threads would be launched to possibly correct a small percentage of inverted triangles), is a way to guarantee the correctness of a simulation in scenarios where multiple edge crossings are more likely to happen.

Acknowledgments

The authors would like to thank the NVIDIA GPU Research Center of the Department of Computer Science of the Universidad de Chile for supplying the equipment used for the tests presented here. This work was partially supported by the FONDECYT projects No. 1140778 and No. 3160182, and by project No. ENL009/15, VID, Universidad de Chile.

References

[1] W.B. Russel, D.A. Saville, W.R. Schowalter, Colloidal Dispersions, Cambridge university press, 1989.
[2] T. Vicsek, A. Zafeiris, Phys. Rep. 517 (3) (2012) 71–140.
[3] M.C. Marchetti, J.F. Joanny, S. Ramaswamy, T.B. Liverpool, J. Prost, M. Rao, R.A. Simha, Rev. Modern Phys. 85 (3) (2013) 1143.
[4] P. Romanczuk, M. Bär, W. Ebeling, B. Lindner, L. Schimansky-Geier, Eur. Phys. J. Spec. Top. 202 (1) (2012) 1–162.
[5] J.L. Anderson, Annu. Rev. Fluid Mech 21 (1) (1989) 61–99.
[6] C.A. Navarro, N. Hitschfeld-Kahler, L. Mateu, Commun. Comput. Phys. 15 (2) (2014) 285–329.
[7] B. Andreotti, Y. Forterre, O. Pouliquen, Granular Media: Between Fluid and Solid, Cambridge University Press, 2013.
[8] T. Pöschel, T. Schwager, Computational Granular Dynamics: Models and Algorithms, Springer, 2005.
[9] M. Weigel, J. Comput. Phys. 231 (8) (2012) 3064–3082.
[10] M. Weigel, Comput. Phys. Comm. 182 (9) (2011) 1833–1836.
[11] J. Bédorf, E. Gaburov, S. Portegies Zwart, J. Comput. Phys. 231 (7) (2012) 2825–2839.
[12] R. Soto, R. Golestanian, Phys. Rev. Lett. 112 (6) (2014) 068301.
[13] R. Soto, R. Golestanian, Phys. Rev. E 91 (5) (2015) 052304.
[14] J. Barnes, P. Hut, Nature 324 (6096) (1986) 446–449.
[15] Z. Yao, J.S. Wang, G.R. Liu, M. Cheng, Comput. Phys. Comm. 161 (1–2) (2004) 27–35.
[16] M.P. Allen, D.J. Tildesley, Computer Simulation of Liquids, Oxford university press, 1989.
[17] D. Frenkel, B. Smit, Understanding Molecular Simulation: From Algorithms to Applications, Vol.1, Academic press, 2001.
[18] M.S. Miguel, R. Toral, in: Instabilities and Nonequilibrium Structures VI, vol. 5, 2000, pp. 35–120.
[19] L. Verlet, Phys. Rev. 159 (1) (1967) 98–103.
[20] T.J. Lipscomb, S.S. Cho, in: ACM Conference on Bioinformatics, Computational Biology and Biomedicine, 2012. pp. 321–328.
[21] A.J. Proctor, T.J. Lipscomb, A. Zou, J.A. Anderson, S.S. Cho, in: Proceedings of the 2012 ASE International Conference on BioMedical Computing, BioMedCom 2012, 2013. pp. 14–19.
[22] L. Nyland, M. Harris, J. Prins, Simulation 3 (1) (2007) 677–696.
[23] M. Burtscher, K. Pingali, GPU Computing Gems Emerald Edition, 2011, pp. 75–92.
[24] K.A. Hawick, D.P. Playne, in: Conferences in Research and Practice in Information Technology Series, vol. 127, 2012, pp. 13–21.
[25] P. Strating, Phys. Rev. E 59 (2) (1999) 2175–2187.
[26] J.S. Vitter, ACM Trans. Math. Software 11 (1) (1985) 37–57.
[27] M. De Berg, O. Cheong, M. Van Kreveld, M. Overmars, Computational Geometry: Algorithms and Applications, Computational Geometry, Vol. 17, 2008.
[28] NVIDIA, Program. Guides (September) (2015) 1–261.
[29] C. Navarro, N. Hitschfeld, E. Scheihing, Commun. Comput. Inf. Sci. 458 (2014) 36–49.
[30] C.L. Lawson, Discrete Math. 3 (4) (1972) 365–372.
[31] The CGAL Project, CGAL User and Reference Manual, fourth ed., CGAL Editorial Board, 2016.
[32] N. Kruithof, 2D Periodic Triangulations, in: CGAL User and Reference Manual, fourth ed., CGAL Editorial Board, 2016.
[33] NVIDIA, CURAND Library: Programming Guide, Version 7.0, NVIDIA, 2015.
[34] S. Green, NVIDIA whitepaper 6 (2010) 121–128.
[35] Y. Fily, M.C. Marchetti, Phys. Rev. Lett. 108 (23) (2012) 235702.