Computer Physics Communications 183 (2012) 1073–1081

Contents lists available at SciVerse ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

Fast neighbor lists for adaptive-resolution particle simulations

Omar Awile [a,b], Ferit Büyükkeçeci [a,b], Sylvain Reboux [a,b], Ivo F. Sbalzarini [a,b,*]

[a] MOSAIC Group, Institute of Theoretical Computer Science, ETH Zurich, CH-8092 Zurich, Switzerland
[b] Swiss Institute of Bioinformatics, ETH Zurich, CH-8092 Zurich, Switzerland

ARTICLE INFO

Article history:
Received 8 March 2011
Received in revised form 27 October 2011
Accepted 3 January 2012
Available online 5 January 2012

Keywords:
Particle methods
Neighbor lists
Multiresolution simulations
Adaptive-resolution simulations
Verlet lists
Cell lists

ABSTRACT

Particle methods provide a simple yet powerful framework for simulating both discrete and continuous systems either deterministically or stochastically. The inherent adaptivity of particle methods is particularly appealing when simulating multiscale models or systems that develop a wide spectrum of length scales. Evaluating particle–particle interactions using neighbor-finding algorithms such as cell lists or Verlet lists, however, quickly becomes inefficient in adaptive-resolution simulations where the interaction cutoff radius is a function of space. We present a novel adaptive-resolution cell list algorithm and the associated data structures that provide efficient access to the interaction partners of a particle, independent of the (potentially continuous) spectrum of cutoff radii present in a simulation. We characterize the computational cost of the proposed algorithm for a wide range of resolution spans and particle numbers, showing that the present algorithm outperforms conventional uniform-resolution cell lists in most adaptive-resolution settings.

1. Introduction

Simulations using particles are ubiquitous in computational science. Particle methods are able to seamlessly treat both discrete and continuous systems either stochastically or deterministically. In discrete particle methods, particles frequently correspond to real-world entities, such as atoms in molecular dynamics simulations or cars in road traffic simulations. In simulations of continuous systems, particles constitute the material points (Lagrangian tracer points) of the system, which evolve according to their pairwise interactions. Examples include the vortex elements in incompressible fluid mechanics simulations [20]. Particle methods are intuitively easy to understand and applicable also in situations that cannot be described by (differential) equations, e.g., in simulations of biological, social, or financial systems.

The efficient evaluation of pairwise particle–particle interactions is a key component of any particle-based simulation. Formally, a set of N interacting particles defines an N-body problem with a nominal computational cost of $O(N^2)$. In many practical applications, however, the particle–particle interactions have a finite range or are truncated with a certain cutoff radius. This reduces the computational cost to $O(N)$ if each particle can find its interaction partners ("neighbors") in $O(1)$ operations.

For constant cutoff radii, two classic data structures are available to provide fast neighbor lists with $O(1)$ access *per particle*: cell (linked) lists [17] and Verlet lists [28]. A cell (linked) list divides the domain into equisized cubic cells with edge lengths equal to the interaction cutoff radius. Each cell then stores a (linked) list of the indices of all particles inside it. When computing particle–particle interactions, each particle can find its neighbors in $O(1)$ time by searching only over the cell it is in and the immediately adjacent cells. Being in one of the neighboring cells is a necessary condition for any particle to be an interaction partner, but the condition is not sufficient. Cell lists hence are conservative and more interaction partners are considered than actually required (up to $3^4/4\pi \approx 6$ times more for a uniform particle distribution in 3D). This overhead can be avoided at the expense of higher memory consumption when using Verlet lists [28] where each particle stores an explicit list of the indices of all its interaction partners. Verlet lists rely on intermediate cell lists for their efficient construction and they commonly include a safety margin (called "skin") in order to avoid their reconstruction every time any particle has moved. This implies a tradeoff between the number of interactions that are computed in excess and the frequency of rebuilding the Verlet lists. For certain systems, optimal skin thicknesses can be found [7,8,27]. Due to the importance and widespread use of cell and Verlet lists, much work has been done to compare and improve their performance [1,21,13,30,29,19].

One of the key advantages of particle methods is their inherent adaptivity. In discrete systems, particles are only needed where the corresponding objects are present. In continuous systems, the particles naturally follow the flow map, again restricting computation to where it is required. The adaptive dynamics of particles, however, can lead to the formation of dense particle clusters. In the worst case, a cluster that is smaller than the particle–particle

* Corresponding author at: MOSAIC Group, Institute of Theoretical Computer Science, ETH Zurich, CH-8092 Zurich, Switzerland.
E-mail address: ivos@ethz.ch (I.F. Sbalzarini).

0010-4655/$ – see front matter
doi:10.1016/j.cpc.2012.01.003

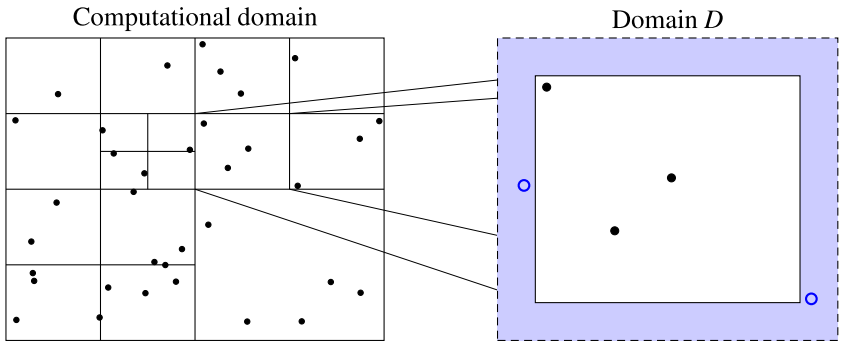

Fig. 1. The computational domain is decomposed into cuboidal subdomains with halo layers (light blue). The halo layers contain ghost particles (blue circles) that are copies of real particles (black dots) from the adjacent subdomains. Independently applying the present algorithm to each extended subdomain (including the halo layers) D allows transparent implementation of boundary conditions and (distributed-memory) parallelism [22]. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

interaction cutoff may contain all the particles. The computational cost of particle methods then deteriorates to $O(N^2)$. This can be avoided by locally adapting the interaction cutoff to the density of particles, leading to *adaptive-resolution* particle methods. Adaptive-resolution methods are required for the efficient simulation of multiscale systems. Hou [18] and Cottet et al. [10] provide two examples of adaptive-resolution particle methods for fluid dynamics; the adaptive-resolution smoothed particle hydrodynamics (SPH) method [26] provides an example from cosmology. In adaptive-resolution simulations the interaction cutoff is defined by a unique-valued map $\mathbf{x} \in \mathbb{R}^d \mapsto r_c(\mathbf{x}) \in \mathbb{R}^+$. This is in contrast to *multi-resolution* simulations where there can be multiple cutoff radii (resolution scales) at any given location. Adaptive-resolution simulations are related to *range-assignment problems* as studied in theoretical computer science, computational geometry, and communication networks [9], where each particle can have a different cutoff radius. If the interaction cutoff is a function of space and hence varies across particles, uniform-resolution cell lists become inefficient and other fast neighbor lists are required.

A number of algorithms and data structures have been proposed to address this or similar problems. *K-d trees* [5] are K-dimensional space-partitioning data structures with a wide range of applications in computational geometry and numerical simulations. They allow efficient k-nearest neighbor searches, but do not support search within a given interaction *radius*. *R-trees* [12,2] relax this constraint by allowing neighborhood searches over bounding boxes. They are prominently used in geographic databases. In 't Veld et al. [19] have proposed multi-resolution cell lists for colloidal mixtures in explicit-solvent molecular dynamics simulations. Their approach assumes a finite number of discrete resolution levels, for each of which a separate uniform-resolution cell list is built.

Here, we present adaptive-resolution cell lists (AR cell lists) that enable efficient access to the neighbors of any particle also in cases where there is a continuous spectrum of interaction cutoff radii, potentially spanning several orders of magnitude. This is achieved by combining cell lists with a tree subdivision of the domain. We present the details of the required data structures and algorithms and demonstrate the construction of AR cell lists and their use to compute particle–particle interactions and to construct the corresponding Verlet lists in adaptive-resolution particle methods.

We benchmark the construction and use of AR cell lists for a wide range of resolution spans and compare them to conventional cell lists. The results show that already in simulations with a modest ratio between the largest and smallest interaction cutoffs, AR cell lists outperform conventional cell lists. AR cell lists enable efficient evaluation of particle–particle interactions also in cases where the cutoff radius varies in space over several orders of magnitude, such as in multiscale and adaptive-resolution particle methods.

2. Adaptive-resolution cell lists

We generalize cell lists to situations where the cutoff radius of the particle–particle interactions is a potentially continuous function of space. Each particle interacts with all other particles within a spherical neighborhood around it. The radius of this neighborhood depends on the location of the center particle. This is most generally modeled by attributing to each particle p its own interaction cutoff radius $r_{c,p}$. We consider the situation where N particles $p = 1, \ldots, N$ are distributed in a cuboidal domain. Boundary conditions and parallelism are handled by decomposing the computational domain into subdomains and extending each subdomain with a halo layer as illustrated in Fig. 1 [23,22]. In a parallel domain-decomposition setting, N hence is the number of particles on the local processor. Since the interaction cutoff locally changes, the halo layers on different sides of a subdomain may have different widths. Populating the halo layers with ghost particles that are copies of real particles from the adjacent subdomains, and treating boundary conditions by imposing specific values on the ghost particles, is assumed to be done prior to AR cell list construction. This is typically the case in parallelization frameworks such as the PPM library [23,22] or PETSc [4]. In order to evaluate the particle–particle interactions in any subdomain, only particles within that subdomain and its halo layer need to be considered. We thus build a separate AR cell list for each extended (including the halo layers) subdomain, hereafter referred to as "domain" D (dashed box in Fig. 1).

Each particle is defined by its position $\mathbf{x}_p \in \mathbb{R}^d$ (for $d = 2$ or 3) and its interaction cutoff radius $r_{c,p} = r_c(\mathbf{x}_p) \in \mathbb{R}^+$. The cutoff radii of neighboring particles may differ by several orders of magnitude and they can take values in a continuum. Two particles are considered neighbors (and hence interact) if

$$\|\mathbf{x}_p - \mathbf{x}_q\| \leqslant \min(r_{c,p}, r_{c,q}), \tag{1}$$

that is, if both are within the interaction radius of the respective other. Following the nomenclature of Hernquist and Katz [14], this neighborhood condition defines a *gather*-type sampling of a particle's neighborhood. For *scatter* interactions, the right-hand side in Eq. (1) would be replaced with $\max(r_{c,p}, r_{c,q})$, and for collision detection with $r_{c,p} + r_{c,q}$ [11]. However, we do not consider these two alternative cases since they may require different data structures than the ones presented here.

In AR cell lists, regions containing particles with small cutoff radii ("small particles") are subdivided into small cells, while regions containing particles with large cutoff radii ("large particles")

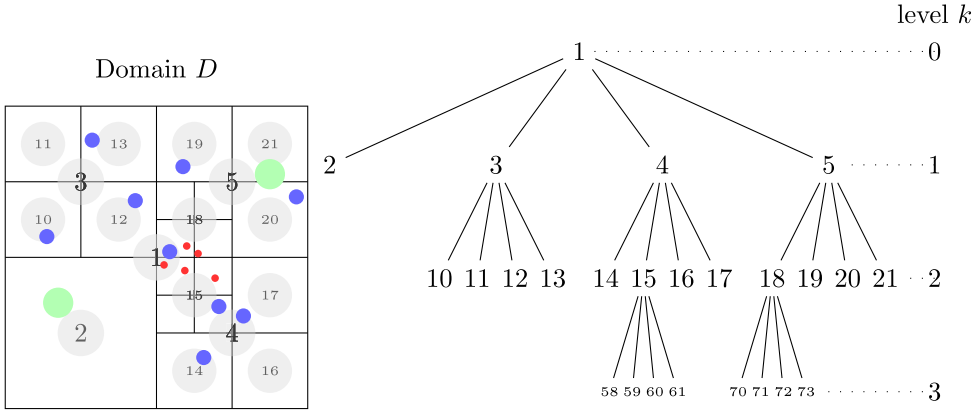

Fig. 2. Left: sketch of an AR cell list with large (green), medium (blue), and small (red) particles. The domain D is adaptively subdivided (black lines). Right: The corresponding cell tree with cells c_k on levels k and level-order indices $J(c_k)$, corresponding to the numbers given in the gray circles in the left panel. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

are subdivided into large cells. These cells are defined as the leafs of an adaptive tree (quad-tree in 2D, oct-tree in 3D). Starting from the entire domain D as the root box of the tree, a tree node is subdivided if it contains particles with a cutoff radius smaller than half the edge length of the cell associated with this node (see Fig. 2, left panel). The association of particles to cells is computed using an in-place Quicksort-like algorithm. The tree nodes are numbered consecutively per level. Numbers corresponding to empty nodes are skipped (see Fig. 2, right panel). This *level-order indexing* of the cell-tree nodes assigns to each tree cell c a unique index $J(c)$ from which it is possible to compute the indices of its neighbor, parent, and child cells in $O(1)$ operations. The resulting cell tree is not stored explicitly, but computed on demand from the particle positions and their levels in the tree.

2.1. Constructing AR cell lists

Standard cell lists organize the particles spatially by sorting them into the cells of a uniform Cartesian mesh. In AR cell lists we additionally organize the particles with respect to their cutoff radii using an adaptive tree data structure. A particle's cutoff radius directly relates to the tree level to which the particle is assigned. The construction of AR cell lists is summarized in Algorithm 1.

Algorithm 1 Constructing AR cell lists in d dimensions.

INPUT: particles $p = 1, \ldots, N$ with positions $\mathbf{x}_p$ and interaction cutoff radii $r_{c,p}$; cuboidal domain D with edge lengths $(D_1, \ldots, D_d)$; $D_m = \min_{i=1,\ldots,d}(D_i)$
OUTPUT: `cells` lookup table containing the cell indices and indices of the first and last particles in each cell.

1. sort particles in descending order by $r_{c,p}$
2. maxlevel $= \lceil \log_2(D_m / \min_p(r_{c,p})) \rceil$
3. assign particles to cell-tree levels:
   A particle with cutoff radius $r_{c,p}$ is assigned to level k, where $D_m/2^k > r_{c,p} \geqslant D_m/2^{k+1}$, $k = 0, \ldots,$ maxlevel $- 1$
4. **for** $k = 0, \ldots,$ maxlevel $- 1$
   (a) partition particles p_k in level k using Algorithm 2.
   Start the recursion of Algorithm 2 with arguments $p = p_k$, $c = D$, `curr_level`$= 1$, and `target_level`$= k$.
   (b) insert the indices of the first and last particle in each leaf of the partitioning into `cells`. Empty leafs are not added; the cell indices in `cells` are hence not contiguous.

This algorithm has two phases:

Phase I. The particles are sorted in order of descending cutoff radii. As this simply amounts to sorting with respect to a scalar property, any efficient sorting algorithm can be used. After the particles have been sorted we determine the tree level each particle belongs to. This starts by computing the level k of the first particle such that

$$D_m/2^k > r_{c,1} \geqslant D_m/2^{k+1}, \quad k = 0, \ldots, \text{maxlevel} - 1. \tag{2}$$

D_m is the minimum edge length of the domain.[1] Subsequently, we linearly iterate through all particle radii $r_{c,p}$, $p = 2, \ldots, N$ and increment k by one whenever $r_{c,p} < D_m/2^{k+1}$.

Phase II. After all particles have been assigned to their respective cell-tree levels we also sort them with respect to their spatial location. This is done using a recursive divide-and-conquer algorithm (Algorithm 2) analogous to Quicksort [16]. In each recursion of the algorithm we are given a set of particles located in the bounding box of a certain tree cell. We first determine the center of the tree cell, $\mathbf{m}$. We then use $\mathbf{m}$ to partition the set of particles in that cell along each dimension into 4 (in 2D) or 8 (in 3D) subsets. This is done by successively using the ith component, $i = 1, \ldots, d$, of $\mathbf{m}$ as the respective pivot and $\geqslant$ as the comparison operator. The same partitioning procedure is then recursively applied to each of the resulting subsets in their respective sub-cells. The recursion stops after k iterations for all particles living on tree level k. The partitioning recursion is separately done for each non-empty tree level, always starting from the entire domain D. This causes the particles on each level to sift down to their respective leafs, starting from the root of the tree.

After Phase II, the particle array is partitioned both by tree levels and by particle positions. Furthermore, the position sorting procedure returns all pairs of indices of the first and last particle in each cell. This information is stored in a lookup table such that the particles belonging to a certain cell can be found in $O(1)$ operations.

2.2. Operations on AR cell lists

Once the AR cell lists are built, a number of operations on them are required in order to efficiently compute particle–particle interactions or construct Verlet lists. These operations are:

Op1: Finding a cell

The cell c_k in which a position $\mathbf{x}$ and cutoff radius r_c is located can be determined by first computing the level in the cell tree as

$$k = \lceil \log_2(D_m/r_c) \rceil$$

[1] In practice we first render the domain cubic by extending it in all directions to its maximum edge length. This avoids constraining the tree depth by the domain's aspect ratio.

Algorithm 2 Sorting the particles by their position.

INPUT: particles p with positions $\mathbf{x}_p$; cell c in which these particles live; the level to be partitioned in this recursion `curr_level`; the level on which the particles p live `target_level`.
OUTPUT: the sorted particle array and the indices of the first and last particle in that array belonging to the cell c.

1. compute the center $\mathbf{m} = (m_1, \ldots, m_d)$ of the cell c and the bounds of the equisized subcells $c_1, \ldots, c_{2^d}$
2. set initial partition to contain all particles, $P_1 = \{p\}$, and initial set of partitions $S = \{P_1\}$
3. **for** $i = 1, \ldots, d$
   (a) **for** $j = 1, \ldots, 2^{i-1}$
      i. partition P_j along m_i into $P_j^{<m_i} = \{p\colon x_{p,i} < m_i\}$ and $P_j^{\geqslant m_i} = \{p\colon x_{p,i} \geqslant m_i\}$
      ii. replace P_j in S with $P_j^{<m_i}, P_j^{\geqslant m_i}$
   The resulting partitioning divides the particles into 2^d disjoint sets $\{p\colon \mathbf{x}_p \in c_i\}$, $i = 1, \ldots, 2^d$
4. **if** `curr_level == target_level`
   (a) **return**
5. **else**
   (a) **for** $i = 1, \ldots, 2^d$
      i. `Algorithm2(`$\{p : \mathbf{x}_p \in c_i\}$`,` c_i`, curr_level+1, target_level)`

and then traversing the tree from its root to level k. During traversal we check for each tree node in which of its quadrants (in 2D) or octants (in 3D) $\mathbf{x}$ is located and descend into the respective child node to locate c_k.

Op2: Finding all particles in a cell

Given a cell index, we can look up the index of the first and last particle inside that cell in the `cells` table. Since not all cell indices exist, this can be done in $O(1)$ time by implementing `cells` as a hash table with the cell index as its key and the pair of particle indices as its value.

Op3: Finding the child cells of a cell

The indices of the children of a cell c are given by $J(c) \cdot 2^d + l$, $l = -2^d + 2, \ldots, 1$.

Op4: Finding the parent cell

The index of the parent cell of a cell c is $\lfloor (J(c) + 2^d - 2)/2^d \rfloor$.

Op5: Finding neighboring cells

The neighbor cells of a cell c are found by adding/subtracting the cell-edge length to/from the center $\mathbf{m}$ of cell c and using these locations $\mathbf{x}$ and the cutoff radius r_c of the tree level of cell c as arguments to Op1. If a neighbor cell does not exist in the `cells` data structure, this means that there are no particles in its region on this level and below, or that the requested cell lies outside the domain.

2.3. Using AR cell lists

Using the AR cell list data structures and the above-defined five operations, every particle can efficiently find all other particles within its neighborhood. This is done by retrieving for each particle all particles in the same cell, in all neighboring cells, and in all descendent cells of the cell tree.

This can also be used to efficiently construct Verlet lists [28] in adaptive-resolution particle simulations. A Verlet list is a data structure that explicitly stores the interaction partners of each particle, allowing each particle to directly access its neighbors. This further reduces the overhead compared to directly using AR cell lists for computing the particle–particle interactions, provided the Verlet lists do not need to be reconstructed at each time step of a simulation. In order to ensure this, the cutoff radius of each particle is enlarged by a safety margin, called "skin". The Verlet lists then only need to be reconstructed once any particle has moved further than its skin thickness.

Evaluating particle–particle interactions or constructing Verlet list based on AR cell lists starts from the particles living on the highest (coarsest) non-empty level of the cell tree and then proceeds level by level. It is therefore convenient to iterate through the particle array in the order given by the sorting produced by Algorithms 1 and 2.

We refer to interactions as *symmetric* when an interaction between particle p and q implies the same (possibly with negative sign) interaction between q and p. This symmetry can be exploited when evaluating particle interactions in order to avoid redundant calculations.

For each particle p we use *Op1* to determine the cell c_k in which it lives and then retrieve the $3^d - 1$ neighboring cells using *Op5*. When building Verlet lists or evaluating *symmetric* interactions, we only need to find one partner in every interaction *pair*. This is illustrated in Fig. 3. We use *Op2* to loop over particles q in c_k. For symmetric interactions or when building Verlet lists, this loop only considers particles in c_k with an index $> p$. Subsequently, we loop over all particles q in the neighboring cells. For symmetric interactions and when building Verlet lists it is sufficient to consider only those neighbors of c_k with an index $> J(c_k)$. For each pair (p, q) we check whether the particles fulfill Eq. (1). Depending on whether the particle interactions are symmetric or not, only q is added to the Verlet list of p, or the two interaction partners are mutually added to each other's lists.

We then recursively use *Op3* and *Op5* to visit all descendent cells of c_k and their respective neighbors. We consider only those descendent cells c_k^δ of c_k on all finer levels $\delta = k+1, \ldots,$ maxlevel that contain the position $\mathbf{x}_p$. Since the cell tree is not stored explicitly, but computed on demand, we first determine the positions and edge lengths of c_k^δ and its neighboring cells $\mathcal{N}(c_k^\delta)$. Note that because we are iterating from large to small particles, we have to visit all $3^d - 1$ neighbors of c_k^δ in order to find all neighboring particles of p on higher levels of resolution, irrespective of whether the interactions are symmetric or not. We then retrieve all particles q in $c_k^\delta \cup \mathcal{N}(c_k^\delta)$ and check whether they fulfill Eq. (1) with particle p. Those that fulfill Eq. (1) are added to the Verlet list of p (and vice versa for asymmetric interactions), or their interactions with p are computed.

The complete procedure for computing particle–particle interactions or building Verlet lists based on AR cell lists is summarized in Algorithm 3. Note that even though *Op4* is not used here, it would be necessary if one were to compute asymmetric particle–particle interactions directly based on AR cell lists, i.e., without building Verlet lists. We do not consider this case here.

2.3.1. Special treatment of halo layers for symmetric particle interactions

In a domain-decomposition setting, the present AR cell list algorithm operates independently on each subdomain of the computational domain (see Figs. 1 and 4). We rely on prior domain decomposition and population of the halo layers by the software in which the algorithm is embedded. This can also directly account for periodic boundary conditions, as also illustrated in Fig. 4. A parallel implementation of Algorithms 1 to 3 is hence not required. If the particle interactions are symmetric, halo layers are only needed on half of the (sub-)domain faces, halving the communication volume. This is illustrated in Fig. 4b. Since the interaction cutoff locally changes, the halo layers on different sides of a (sub-)domain may have different widths. Symmetric particle interaction schemes also change the properties (values) of ghost particles. These ghost contributions then have to be sent back to the corresponding real particle and properly accounted for (for example using the `ghost_put` mapping of the PPM library [23]). Sym-

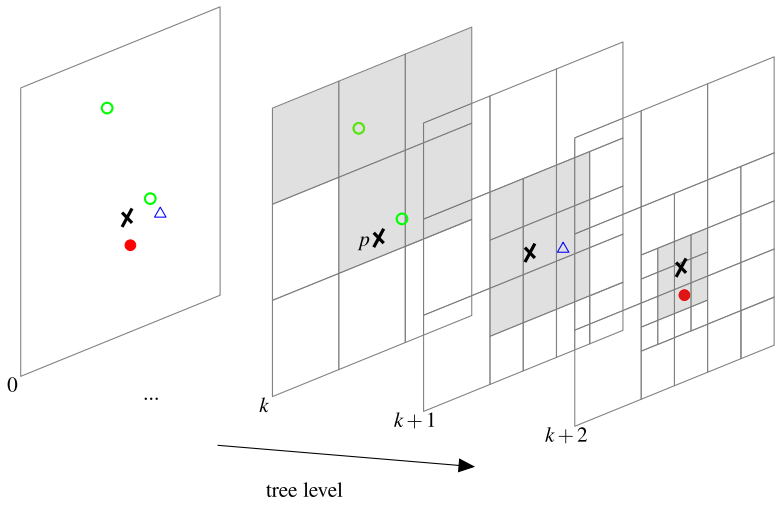

Fig. 3. Finding interaction partners of a particle in an AR cell list (one iteration of Algorithm 3). The back plane (tree level 0) shows all particles without the cell-tree decomposition. In order to compute a symmetric interaction (or construct the Verlet list) of particle p (black cross) on level k we first iterate through all particles (green circles) in half of the neighboring cells $\mathcal{N}(c_k)$ on the same tree level (shaded cells on level k). Then, the finer tree levels are searched for interaction partners in the descendent cells c_k^δ and their neighbors $\mathcal{N}(c_k^\delta)$ (shaded cells on levels $k+1$ and $k+2$) on all finer levels $\delta = k+1, \ldots,$ maxlevel. On these finer levels, all neighboring cells must be visited in order to include all interaction pairs across different levels of resolution (blue triangle on level $k+1$ and red dot on level $k+2$). Transparent cells are not considered when computing this interaction. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Algorithm 3 Computing particle–particle interactions or building Verlet lists based on AR cell lists.

INPUT: particles $p = 1, \ldots, N$ with positions $\mathbf{x}_p$ and cutoff radii $r_{c,p}$.
OUTPUT: result of the particle–particle interaction or Verlet list storing for each particle the indices of all particles within its neighborhood.

for each particle p

1. determine the cell c_k containing p ($\mathbf{x}_p$, $r_{c,p}$) using (*Op1*).
2. **if** computing symmetric particle–particle interactions or constructing Verlet lists **then**
   (a) retrieve those neighbors of c_k with index $> J(c_k)$, $\mathcal{N}(c_k)$, using (*Op5*).
   (b) **for** each particle $q > p \in c_k$ and each particle $q \in \mathcal{N}(c_k)$ (*Op2*)
      i. **if** $\|\mathbf{x}_p - \mathbf{x}_q\| \leqslant \min(r_{c,p}, r_{c,q})$ **then** add q to the Verlet list of p (and vice versa when later computing asymmetric interactions based on these Verlet lists), or compute the interaction between particles p and q.
3. **else**
   (a) retrieve all neighbors of c_k, $\mathcal{N}(c_k)$, using (*Op5*).
   (b) **for** each particle $q \in (c_k \cup \mathcal{N}(c_k))$ (*Op2*)
      i. **if** $\|\mathbf{x}_p - \mathbf{x}_q\| \leqslant \min(r_{c,p}, r_{c,q})$ **then** compute the interaction between particles p and q.
4. **for** $\delta = k+1, \ldots,$ maxlevel
   (a) use (*Op3*) to determine the cell c_k^δ that is the $(k-\delta)^{\text{th}}$ descendant of c_k and contains the location $\mathbf{x}_p$.
   (b) retrieve all neighbors of c_k^δ, $\mathcal{N}(c_k^\delta)$, using (*Op5*)
   (c) **for** each particle $q \in (c_k^\delta \cup \mathcal{N}(c_k^\delta))$ (*Op2*)
      i. **if** $\|\mathbf{x}_p - \mathbf{x}_q\| \leqslant \min(r_{c,p}, r_{c,q})$ **then** add q to Verlet list of p (and vice versa when later computing asymmetric interactions based on these Verlet lists), or compute the interaction between particles p and q.

metric interactions can additionally result in two ghost particles interacting. These ghost–ghost interactions are efficiently found using bitwise operations as follows: Each ghost particle is assigned a d-bit string where the ith bit is 1 if the particle is in the halo layer in dimension k and 0 otherwise. If a bitwise AND operation on the bit strings of two ghost particles results in 0, these ghosts interact.

3. Results

We implemented Algorithms 1 through 3 in Fortran 90 and performed several computer experiments to benchmark their computational efficiency and evaluate the performance gain over uniform-resolution cell lists as a function of the spectrum of scales spanned by the cutoff radii and of the total number of particles in the domain. In all benchmarks, we verified that the AR cell lists found the correct set of interactions. The reference implementations of uniform-resolution cell and Verlet lists were taken from the PPM library [23,3] and are also implemented in Fortran 90. All benchmark codes were compiled using the Intel Fortran compiler version 12.0 with the `-O3` optimization flag. The benchmarks were run on a 2.8 GHz Intel Xeon E5462 CPU.

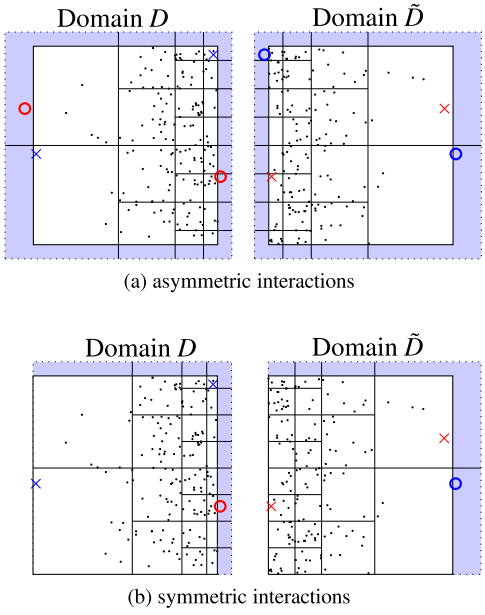

Fig. 4. Halo layers for symmetric and asymmetric neighbor lists and treatment of periodic boundary conditions. The computational domain is decomposed into two (sub-)domains D and $\tilde{D}$ (cf. Fig. 1). On each (sub-)domain and its respective halo layers, a separate AR cell tree (black lines) is built. Blue crosses indicate particles in domain D that are ghosts in the halo layer of domain $\tilde{D}$ (blue circles). The red crosses highlight two particles from domain $\tilde{D}$ that are ghosts on domain D (red circles). For each color, two examples are shown: one for periodic boundary conditions, the other for internal (sub-)domain boundaries. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3.1. Benchmarks

We measure the computational time for building and using AR cell lists over different particle distributions. In all distributions we

place a fixed number of 10×10 particles on a uniform Cartesian mesh with spacing $h_b = 0.1$ and set their interaction radii $r_{c,b} = 3h_b/2$. For each distribution we then choose a *resolution span* $\lambda = \max_p(r_{c,p})/\min_p(r_{c,p})$ and a number of small particles N. These additional small particles are given interaction radii $r_{c,s} = r_{c,b}/\lambda$ and are placed on a uniform Cartesian mesh with spacing $h_s = 2r_{c,s}/3$ adjacent to the coarse mesh. Fig. 5 shows an example of a resulting adaptive-resolution particle distribution. Similar particle distributions may arise in simulations of shock waves in compressible fluids. For the present benchmarks, the interaction radii are chosen such that each particle always has exactly 8 interaction partners, which allows comparing timing results across resolution spans.

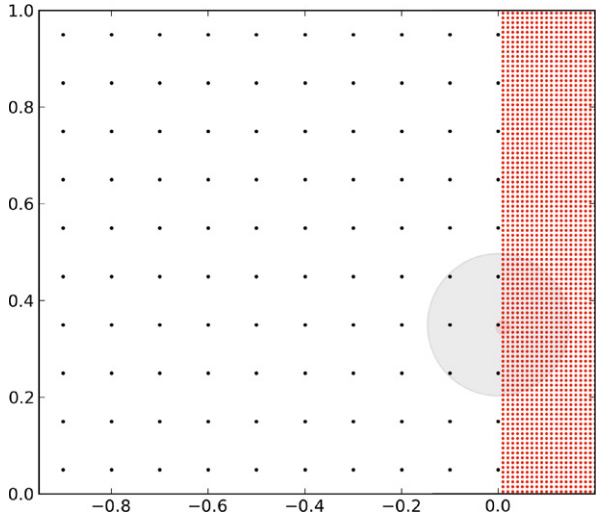

Fig. 5. An example particle distribution used for the present benchmarks. In this figure $N = 2000$ and $\lambda = 10$. The "large" black particles have a cutoff radius of 0.15, while the "small" red particles have a cutoff radius of 0.015 ($\lambda = 0.15/0.015 = 10$). For comparison, the interaction ranges of two neighboring particles at the resolution interface are shown as shaded circles of the respective color. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

We first measure the runtime scaling for constructing AR cell list and conventional cell list for increasing numbers of particles and constant λ. We repeat this experiment for $\lambda = [1, 10, 100, 1000]$ to cover a wide range of resolution spans. The results are shown in Fig. 6. Constructing AR cell lists is about one order of magnitude slower than constructing conventional cell lists. A quick analysis of Algorithm 1 shows that Step 1 can be accomplished in $O(N \log N)$ time. Step 2 can directly be computed in $O(1)$. Step 3 is essentially a linear iteration through the N particles and therefore has a runtime of $O(N)$. Step 4 linearly depends on the number of cell tree levels, which in turn depends on λ. If the number of interaction partners of each particle is bounded by a constant, the overall runtime of the algorithm is $O(\text{maxlevel} \times N \log N)$. This is a higher computational complexity than the $O(N)$ runtime for building conventional cell lists.

Fig. 6 also shows the total runtimes to construct conventional and AR cell lists *and* build Verlet lists based on them for $\lambda = [1, 10, 100, 1000]$. For $\lambda = 1000$ and 10^6 particles (Fig. 6(d)), building the cell lists and the Verlet lists for all particles is almost three orders of magnitude faster when using AR cell lists instead of conventional ones. Figs. 6(b) and (c) further show that the runtime of constructing Verlet lists based on conventional cell lists is first $O(N^2)$ and then decreases to $O(N)$ beyond a "saturation point". This can be understood as follows: Since the cells of conventional cell lists are as large as the largest cutoff radius in the domain, the runtime of the particle–particle interactions using conventional cell lists in the present test case is about $100 + N^2/N_{\text{cells}}$ for 100 particles with large $r_{c,b}$ and N particles with small $r_{c,s}$. For small N the total number of cells in the cell list, N_{cells}, is constant and the quadratic term dominates, leading to a quadratic runtime as more

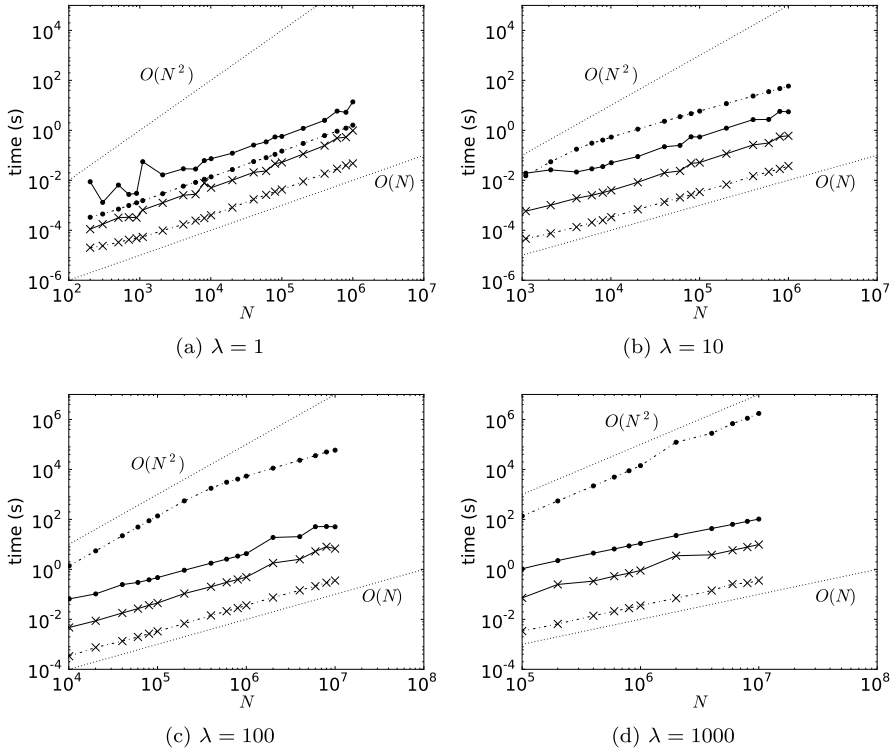

Fig. 6. Runtime for increasing numbers of particles N and resolution spans $\lambda = [1, 10, 100, 1000]$, (a)–(d). Each plot shows the total runtime for constructing (crosses) conventional (dashed lines) and AR (solid lines) cell lists in 2D and for constructing the cell lists *plus* constructing Verlet lists based on them (dots). The theoretical slopes of an $O(N)$ and an $O(N^2)$ algorithm are indicated by the dotted lines.

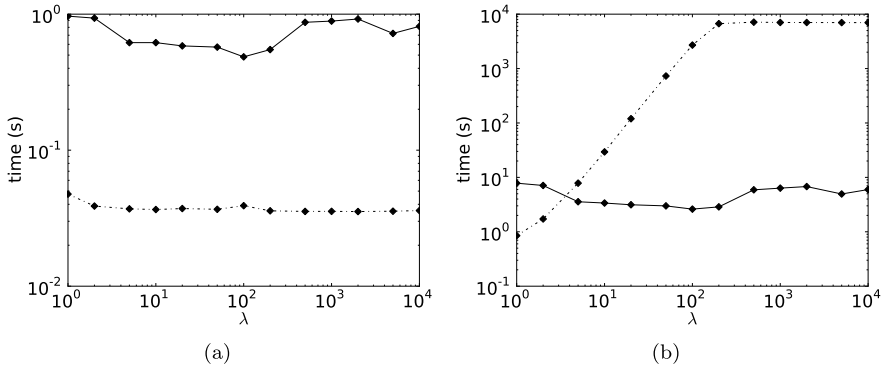

Fig. 7. Total runtime for increasing resolution spans $1 \leqslant \lambda \leqslant 10\,000$ and constant $N = 10^6$. (a) Runtime for constructing conventional (dashed line) and AR (solid line) cell lists in 2D. (b) Total runtime for constructing conventional (dashed line) and AR (solid line) cell list *and* constructing Verlet list based on them.

and more small particles are added into the constant number of cells covering the domain. For large-enough N, after the rightmost column of cells has been completely filled with small particles, N_{cells} increases proportionally with N, rendering the runtime linear beyond this saturation point. This can be seen in Figs. 6(b) and (c) as a reduction in the slope of the particle–particle interaction runtime curve. As λ increases, the saturation point shifts to larger N.

The runtimes of AR and conventional cell lists depend on the spectrum of scales λ present in the particle distribution. For $\lambda = 1$ conventional cell lists are more efficient (see Fig. 6(a)). For increasing λ, the additional overhead for constructing the AR cell lists is gradually amortized by their higher efficiency when computing particle–particle interactions. We therefore repeat the benchmarks for different values of λ between 1 and 10^4 and measure the total runtime. The measured runtimes are shown in Fig. 7. As expected, the cost of constructing conventional cell lists is independent of λ and about one order of magnitude lower than for the AR variant (Fig. 7(a)). When using AR cell lists to build Verlet lists, however, the computational cost is virtually independent of λ, whereas for conventional cell lists it rapidly grows with λ (Fig. 7(b)). This is expected as the particles cluster more and more and the average number of particles per cell grows (quadratically in 2D and cubically in 3D) for conventional cell lists, whereas it remains constant in AR cell lists. The runtime for building the Verlet lists using conventional cell lists reaches a plateau at $\lambda = 200$. This can be explained by the specific arrangement of particles used in the present benchmark. At $\lambda > 200$ the particles with small cutoff radii are so tightly arranged that they all fit into the minimum number of cells required to cover the interface between the large and small particles.

We determine the break-even value of λ where the overall runtime for constructing AR cell lists and using them to construct Verlet lists drops below that for constructing conventional cell lists and building Verlet lists based on them. For $\lambda = 1$, constructing Verlet lists form conventional cell lists is about 25% faster than constructing them from AR cell lists. Already for $\lambda = 3.65$, however, the overall runtime for AR cell lists is equal to that for conventional cell lists. For resolution spans of about $\lambda = 10$, AR cell lists are about one order of magnitude faster than conventional ones. This indicates that the use of AR cell lists is advantageous in most adaptive-resolution particle simulations, even for modest resolution spans.

3.2. Example application

As an example application where AR neighbor lists may be advantageous we consider diffusion on a curved surface simulated using an adaptive-resolution variant of a smooth particle method [6]. The surface is represented implicitly as a level set [25] that is discretized using particles as collocation points [15]. Diffusion amounts to interactions between neighboring particles as defined by DC-PSE operators [24].

We consider a surface of revolution generated by three arcs of circles, resembling a small bud pinching off from a larger sphere (see Fig. 8). This models the geometry of a dividing yeast cell. The radii of the bud and of the sphere are fixed to 1 and 3, respectively. The radius of curvature at the neck, r_P, is varied parametrically in order to tune the resolution span present in the problem.

In order to properly resolve the geometry, the density of particles needs to be larger (and their interaction radii smaller) in regions where the surface has a large curvature. We hence place the particles such that the distance between neighboring particles is proportional to the local radius of curvature of the surface. The cutoff radii hence span a continuous spectrum of scales and the geometry is well resolved everywhere, as shown in Fig. 8. Particles are only placed in a narrow band around the surface and the rest of the volume remains empty [6]. Varying the neck curvature r_P leads to different ratios between the largest and the smallest curvature of the surface, and hence to different resolution spans λ. The mean resolution h_0 on the larger sphere is fixed in each run, so that decreasing r_P (i.e., increasing λ) leads to an increase in the total number of particles N.

We measure the computational cost of constructing and using the cell lists using the present AR method and compare it to the cost of conventional cell lists for mean resolutions $h_0 = [0.1, 0.2, 0.45]$ and λ varying between 3 and 2000. Fig. 9(a) shows the total runtime for constructing the Verlet lists using either AR cell lists or conventional cell lists. For the coarsest resolution, the break-even point is around $\lambda = 60$. This reduces to $\lambda = 4$ for $h_0 = 0.2$ and to $\lambda < 2$ for the finest resolution considered. Fig. 9(b) shows the runtime per particle for constructing the cell and Verlet lists, demonstrating that AR cell lists provide neighbor access with a runtime that is insensitive to the resolution span λ and to the total number of particles N as realized by the different resolutions. This is in contrast to conventional cell lists whose runtime significantly increases with λ and with increasing N (decreasing h_0).

4. Conclusions

We have presented data structures and algorithms for efficiently finding the interaction partners of each particle in a particle-based simulation with short-range interactions whose cutoff radii vary between particles. This enables efficient computation of limited-range particle–particle interactions in adaptive-resolution simulations with a potentially continuous spectrum of cutoff radii. Constructing adaptive-resolution (AR) neighbor lists is computationally more expensive than constructing conventional uniform-resolution neighbor lists. This additional overhead, how-

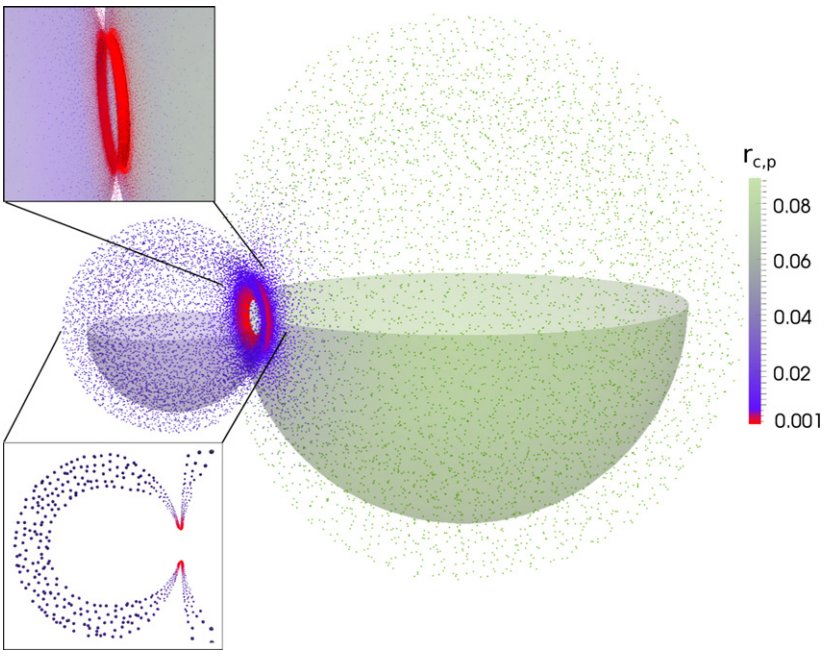

Fig. 8. Particle distribution used in the present example of an adaptive-resolution simulation of diffusion on a surface. The surface is axially symmetric and only its lower half is shown. The surface is represented as a level set discretized on the particles and restricted to a narrow band. Both the width of the narrow band and the cutoff radii of the particles, represented by the color code, depend on the local surface curvature. The high curvature at the neck between the two sphere shells requires a locally increased resolution. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

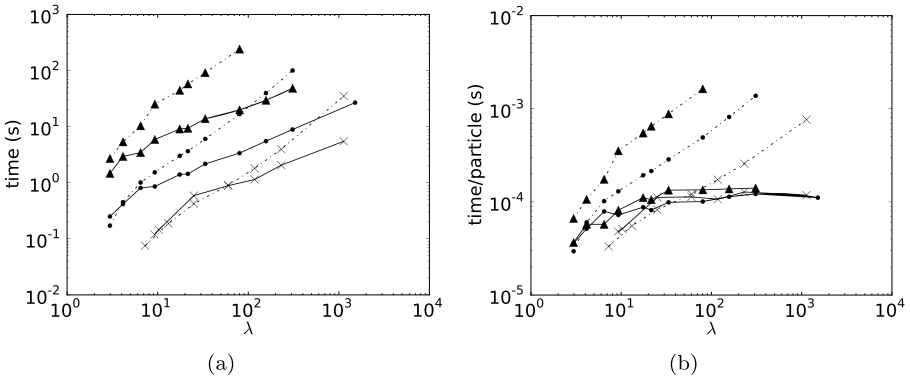

Fig. 9. Runtime for constructing the cell lists *and* using them to construct Verlet lists for the test case shown in Fig. 8. For both conventional (dashed lines) and AR (solid lines) cell lists we vary the resolution span λ and the mean resolution h_0 on the larger sphere, hence varying the total number of particles N. Crosses, dots, and triangles correspond to $h_0 = 0.1$, 0.2, and 0.45, respectively. Note that N increases with λ. The two panels show: (a) the total runtime and (b) the runtime per particle.

ever, is quickly amortized by the gain in performance when using the AR cell lists to evaluate particle–particle interactions or to construct Verlet lists for adaptive-resolution particle distributions. Already at modest ratios between the cutoff radii of the largest and smallest particles in a simulation AR cell lists are faster overall. The actual break-even point, however, depends on the specific particle distribution. The larger the spectrum of scales that are present in a simulation, the bigger the computational saving becomes. For realistic adaptive-resolution simulations, the present AR cell lists can be several orders of magnitude faster than conventional cell lists.

We have implemented both AR cell lists and Verlet lists based on AR cell lists in the PPM library [23,3] in order to make them available for adaptive-resolution simulations on parallel distributed-memory computers. In PPM, the presented algorithms are applied locally per subdomain (i.e., per processor) of a domain decomposition. They thus have no impact on the communication overhead of a parallel simulation, assuming that the halo layers are populated beforehand. The PPM library provides an application-independent middleware for large-scale parallel hybrid particle-mesh simulations. The library is available free of charge and as open source from http://www.ppm-library.org.

Acknowledgements

S.R. was financed by a grant from the Swiss SystemsX.ch initiative, grant LipidX, evaluated by the Swiss National Science Foundation, to I.F.S. O.A. and F.B. were funded by grant #200021-132064 from the Swiss National Science Foundation, to I.F.S. We thank all members of the MOSAIC Group (ETH Zurich) for many fruitful discussions.

References

[1] M.P. Allen, D.J. Tildesley, Computer Simulation of Liquids, Clarendon Press, Oxford, 1987.

[2] L. Arge, M. de Berg, H.J. Haverkort, K. Yi, The priority R-tree: a practically efficient and worst-case optimal R-tree, in: Proc. SIGMOD, Intl. Conf. Management of Data, ACM, Paris, France, 2004.

[3] O. Awile, O. Demirel, I.F. Sbalzarini, Toward an object-oriented core of the PPM library, in: Proc. ICNAAM, Numerical Analysis and Applied Mathematics, International Conference, AIP, 2010, pp. 1313–1316.
[4] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, B.F. Smith, H. Zhang, PETSc users manual, Technical report ANL-95/11, Revision 3.1, Argonne National Laboratory, 2010.
[5] J.L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM 18 (September 1975) 509–517.
[6] M. Bergdorf, I.F. Sbalzarini, P. Koumoutsakos, A Lagrangian particle method for reaction-diffusion systems on deforming surfaces, J. Math. Biol. 61 (2010) 649–663.
[7] A.A. Chialvo, P.G. Debenedetti, On the use of the Verlet neighbor list in molecuar dynamics, Comput. Phys. Comm. 60 (1990) 215–224.
[8] A.A. Chialvo, P.G. Debenedetti, On the performance of an automated Verlet neighbor list algorithm for large systems on a vector processor, Comput. Phys. Comm. (1991) 15–18.
[9] A. Clementi, P. Crescenzi, P. Penna, G. Rossi, P. Vocca, On the complexity of computing minimum energy consumption broadcast subgraphs, in: Proc. Symp. Theoretical Aspects of Computer Science, 2001, pp. 121–131.
[10] G.-H. Cottet, P. Koumoutsakos, M.L. Ould Salihi, Vortex methods with spatially varying cores, J. Comput. Phys. 162 (1) (2000) 164–185.
[11] C. De Michele, Optimizing event-driven simulations, Comput. Phys. Comm. 182 (9) (2011) 1846–1850. Special edition for Conference on Computational Physics, Trondheim, Norway, June 23–26, 2010.
[12] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proc. SIGMOD, Intl. Conf. Management of Data, ACM, 1984, pp. 47–57.
[13] T.N. Heinz, P.H. Hünenberger, A fast pairlist-construction algorithm for molecular simulations under periodic boundary conditions, J. Comput. Chem. 25 (12) (2004) 1474–1486.
[14] L. Hernquist, N. Katz, TREESPH – a unification of SPH with the hierarchical tree method, Astrophys. J. Suppl. Ser. 70 (June 1989) 419–446.
[15] S.E. Hieber, P. Koumoutsakos, A Lagrangian particle level set method, J. Comput. Phys. 210 (2005) 342–367.
[16] C.A.R. Hoare, Quicksort, Comput. J. 5 (1) (1962) 10–16.
[17] R.W. Hockney, J.W. Eastwood, Computer Simulation using Particles, Institute of Physics Publishing, 1988.
[18] T.Y. Hou, Convergence of a variable blob vortex method for the Euler and Navier–Stokes equations, SIAM J. Numer. Anal. 27 (6) (1990) 1387–1404.
[19] P.J. in't Veld, S.J. Plimpton, G.S. Grest, Accurate and efficient methods for modeling colloidal mixtures in an explicit solvent using molecular dynamics, Comput. Phys. Comm. 179 (5) (2008) 320–329.
[20] P. Koumoutsakos, Multiscale flow simulations using particles, Annu. Rev. Fluid Mech. 37 (2005) 457–487.
[21] W. Mattson, B.M. Rice, Near-neighbor calculations using a modified cell-linked list method, Comput. Phys. Comm. 119 (2–3) (1999) 135–148.
[22] I.F. Sbalzarini, Abstractions and middleware for petascale computing and beyond, Intl. J. Distr. Systems Technol. 1 (2) (2010) 40–56.
[23] I.F. Sbalzarini, J.H. Walther, M. Bergdorf, S.E. Hieber, E.M. Kotsalis, P. Koumoutsakos, PPM – a highly efficient parallel particle-mesh library for the simulation of continuum systems, J. Comput. Phys. 215 (2) (2006) 566–588.
[24] B. Schrader, S. Reboux, I.F. Sbalzarini, Discretization correction of general integral PSE operators in particle methods, J. Comput. Phys. 229 (2010) 4159–4182.
[25] J.A. Sethian, Level Set Methods and Fast Marching Methods, Cambridge University Press, Cambridge, UK, 1999.
[26] P.R. Shapiro, H. Martel, J.V. Villumsen, J.M. Owen, Adaptive smoothed particle hydrodynamics, with application to cosmology: methodology, Astrophys. J. Suppl. Ser. 103 (1996) 269.
[27] G. Sutmann, V. Stegailov, Optimization of neighbor list techniques in liquid matter simulations, J. Mol. Liq. 125 (2006) 197–203.
[28] L. Verlet, Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules, Phys. Rev. 159 (1) (1967) 98–103.
[29] U. Welling, G. Germano, Efficiency of linked cell algorithms, Comput. Phys. Comm. 182 (2011) 611–615.
[30] Z. Yao, J.-S. Wang, G.-R. Liu, M. Cheng, Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method, Comput. Phys. Comm. 161 (1–2) (2004) 27–35.