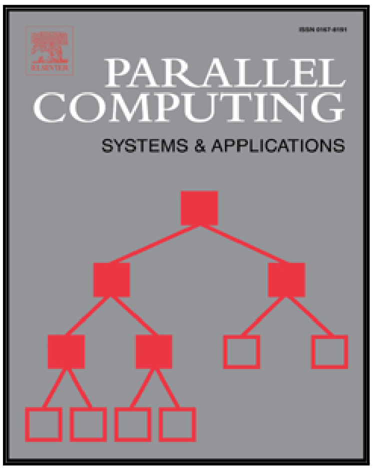
Two particle-in-grid realisations on spacetrees

T. Weinzierl, B. Verleye, Pierre Henri, D. Roose



Two Particle-in-Grid Realisations on Spacetrees

T. Weinzierl, B. Verleye, P. Henri, D. Roose

PII: S0167-8191(15)00163-5
DOI: 10.1016/j.parco.2015.12.007
Reference: PARCO 2296

To appear in: *Parallel Computing*

Received date: 22 January 2015
Revised date: 22 December 2015
Accepted date: 28 December 2015

Please cite this article as: T. Weinzierl, B. Verleye, P. Henri, D. Roose, Two Particle-in-Grid Realisations on Spacetrees, *Parallel Computing* (2016), doi: 10.1016/j.parco.2015.12.007

Highlights

- We introduce two algorithms to hold particles in dynamically adaptive grids.
- Particles are associated to the nearest cell or vertex though they may tunnel.
- A PIC code validates that particle-grid interactions can be realised efficiently.
- Measurements reveal which variant performs better for certain setups and hardware.
- A data movement prediction scheme avoids all-to-all communication partially.

Two Particle-in-Grid Realisations on Spacetrees

T. Weinzierl[a,*], B. Verleye[b,c], P. Henri[d], D. Roose[b]

[a]*School of Engineering and Computing Sciences, Durham University*
Stockton Road, Durham DH1 3LE, Great Britain
[b]*Department of Computer Science, KU Leuven*
Celestijnenlaan 200A, B-3001 Leuven, Belgium
[c]*Vrije Universiteit Brussel*
Pleinlaan 2, 1050 Elsene, Belgium
[d]*Laboratoire de Physique et Chimie de l'Environnement et de l'Espace (LPC2E)*
CNRS, Université d'Orléans
45071 Orléans Cedex 2, France

Abstract

The present paper studies two particle management strategies for dynamically adaptive Cartesian grids at hands of a particle-in-cell code. One holds the particles within the grid cells, the other within the grid vertices. The fundamental challenge for the algorithmic strategies results from the fact that particles may run through the grid without velocity constraints. To facilitate this, we rely on multiscale grid representations. They allow us to lift and drop particles between different spatial resolutions. We call this cell-based strategy particle in tree (PIT). Our second approach assigns particles to vertices describing a dual grid (PIDT) and augments the lifts and drops with multiscale linked cells.

Our experiments validate the two schemes at hands of an electrostatic particle-in-cell code by retrieving the dispersion relation of Langmuir waves in a thermal plasma. They reveal that different particle and grid characteristics favour different realisations. The possibility that particles can tunnel through an arbitrary number of grid cells implies that most data is exchanged between neighbouring ranks, while very few data is transferred non-locally. This constraints the scalability as the code potentially has to realise global communication. We show that the merger of an analysed tree grammar with PIDT allows us to predict particle movements among several levels and to skip parts of this global communication a priori. It is capable to outperform several established implementations based upon trees and/or space-filling curves.

Keywords: Particle-in-cell, spacetree, particle sorting, AMR,
Lagrangian-Eulerian methods, communication-avoiding

[*]Corresponding author

Email addresses: tobias.weinzierl@durham.ac.uk (T. Weinzierl), bverleye@vub.ac.be (B. Verleye), pierre.henri@cnrs-orleans.fr (P. Henri), dirk.roose@cs.kuleuven.be (D. Roose)

URL: www.dur.ac.uk/tobias.weinzierl (T. Weinzierl)

1. Introduction

Lagrangian-Eulerian descriptions of physical phenomena are used by a wide range of applications. They combine the short-range aptitude of particle-based with the long-range capabilities of grid-based approaches. Besides their popularity driven by application needs, particle-grid methods are also popular in supercomputing. They are among the best scaling algorithms today (cf. for example [8, 20, 21]). This scaling mainly relies on two ingredients. On the one hand, particle-particle interactions often are computationally expensive in terms of floating point operations with moderate memory footprint. On the other hand, the particle-grid interaction requires a mapping of particles to the grid. This is a spatial sorting problem. It either is performed infrequently, is computationally cheap as the particles move at most one grid cell at a time, or the grid can be constructed efficiently starting from the particles [21].

The present paper is driven by a plain electrostatic PIC simulation [9, 27] of an unmagnetised plasma. Its computational profile differs from the previous characteristics as it solves a partial differential equation (PDE) on an adaptive Cartesian grid which has to be stored persistently in-between two time steps. At the same time, the particles do not interact with each other—they induce very low computational workload—but may move at very high speed through the grid. Our work focuses on well-suited data structures and algorithms required for such a code within a dynamic adaptive mesh (AMR) environment where the simultion is ran on a distributed memory machine.

The setup raises an algorithmic challenge. While dynamic AMR for PDEs as well as algorithms based upon particle-particle interactions are exhaustively studied, our algorithm requires a fast mapping of particle effects onto the grid and the other way round per time step. This assignment of particles to the grid changes incrementally. Yet, some particles might *tunnel* several cells per time step: no particle is constrained to move at most into a neighbouring cell. In general, the time step size is chosen such that the majority of particles travel at most one cell per time step. This avoids the finite grid instability [2], i.e. numerical, non-physical, heating of the experiment. However, in our application as well as most non-relativistic plasma and gravitation applications, suprathermal particles do exist. Their velocity is not bounded.

The present paper proposes the parallel, locally refined, dynamically adaptive grid to result from a spacetree [29, 32] yielding a Cartesian tessellation. Particles are embedded into the finest tree level. The latter is the adaptive grid hosting the PDE. A multiscale grid traversal with particle-grid updates then can be realised via a simple recursive code mirroring a depth-first search [30].

Two particle realisation variants are studied: we either store the particles within the spacetree leaves (particle in tree; PIT) or within the dual tree (particle in dual tree; PIDT) induced by the spacetree vertices. The latter is similar to a linked-list approach with links on each resolution level. This multiscale linked-list can be deduced on-the-fly, i.e. is not stored explicitly but encoded within the tree's adjacency information. Both particle storage variants render the evaluation of classical compact particle-grid operators straightforward as each particle is assigned to its spatially nearest grid entity anytime. Tunneling is enabled as particles are allowed to move up and down in the whole spacetree. PIDT furthermore can move particles between neighbours. Besides neighbours and parent-child relations, no global adjacency data structure is required. While PIDT induces a runtime overhead and induces a more complex code compared to PIT, the multiscale linked-list nature of PIDT reduces the particle movements up (lift) and down (drop) in the tree, and particles moving along the links can be exchanged asynchronously in-between grid traversals. If we combine PIDT with a simple analysed tree grammar [11] for the particle velocities, we can predict lifts and anticipate drops in whole grid regions. This helps to overcome a fundamental problem. Since tunneling is always possible, we need all-to-all communication in every time step: each rank has to check whether data is to be received from any other rank. This synchronisation introduces inverse weak scaling. The more particles the higher the number of tunnels. The more ranks the more tunnel checks. With a lift prediction, we can locally avoid the all-to-all that we map onto a reduction (reduction-avoiding PIDT; raPIDT). We weaken the rank synchronisation. Rank and process are used here as synonyms. For reasonably big parallel architectures, PIDT and its extension raPIDT thus outperform PIT. Though all three flavours of particle storage support tunneling, their performance is not significantly slower than a classic linked-list approach not allowing particles to tunnel. The multiscale nature of both PIT and PIDT as well as its variant raPIDT further makes the communication pattern, i.e. the sequence and choice of particles communicated via MPI, comprise spatial information. Ranks receiving particles that have to be sorted into a local grid anticipate this presorting and thus can outperform other classic approaches [4, 23, 25].

The remainder is organised as follows. We refer to related work before we introduce the algorithmic steps of electrostatic PIC motivating our algorithmic research (Section 3). In Section 4, we describe our spacetree grid paradigm. Two particle realisation variants storing particles either within the grid cells or within the vertices form the present work's focus (Section 5). Our experimental evaluation starts from a review of the particle movement characteristics before we study the runtime behaviour of the particle storage and sorting schemes as well as the scaling. Finally, we compare our results to three other approaches. This comparison (Section 7) highlights our contribution with respect to the particular application from Section 3 as well as the inevitable cost introduced by the tunneling particles. A summary and outlook in Section 8 close the discussion. The appendix comprises a real-world validation run of our code as well as additional experimental data.

2. Related and used work

Requiring tunneling in combination with a persistent grid holding a PDE solution renders many established grid data structures inappropriate or unsuited.

We distinguish three classes of alternatives: on-the-fly (re-)construction, sorting and linked-list. Reconstructing the assignment for the prescribed grid from scratch per time step [21] is not an option, as the grid is given, holds data and is distributed. Local sorting followed by scattering is not an option (see e.g. [23, 25] and references therein) as the decomposition, i.e. the adaptive mesh, is not known on each individual rank. The two possible modifications are to rely on replicated grids per rank and then to map these grids onto the real PDE grid—this approach is not followed-up here—or to sort locally within the grid and then to scatter those particles that cannot be sorted locally. Such a two-stage approach enriches local meta data information with additional decomposition data (ranks have to know how to scatter the particles) held on each rank, but may run into network congestion (cf. results from [4]) even if we avoid all-to-all collectives [25] and rely on elegant decomposition paradigms such as space-filling curves (SFCs). Finally, standard linked-cell algorithms, cell-based Verlet lists or hypergraph-based approaches [7, 17] are not well-suited. They impose constraints on the velocities of particles, i.e. they can not handle particles that may travel arbitrarily fast. Alternative approaches such as overlapping domain decompositions where the grid holding the particles is replicated on all ranks, schemes where particle and domain decomposition may differ, schemes where the associativity is re-emended only every k steps or approaches tailored to regular grids only are not considered here [5, 10, 17, 19].

For the present challenge, we require an algorithm that updates the grid-particle correlation incrementally. It has to be fast, i.e. anticipate the incremental character, but nevertheless has to support particle escapees. Hereby, the particle distribution follows the grid decomposition: particles contained within a grid cell should reside on the compute node holding the respective grid cell in a non-overlapping domain decomposition sense. For this, we rely on a tree data structure where grid cells ‘point’ to particles. Our approaches materialise ideas of bucket sorts [25]. A tree’s multiscale nature allows us to realise the sorting incrementally, locally and fast without constraints on the particle velocity or the dynamically adaptive grid structure. The omnipresent multiscale nature and the lack of grid constraints make it differ from the `up_down_tree` approach in [4]. Our PIDT variant holds particles within vertices and thus allows cells to push particles into their neighbouring cells directly. No multiscale data is involved. As such it materialises the linked-cells idea where each cells holds a list of its neighbours. As these links are available on all discretisation levels, our PIDT variant can be read as a multiscale linked-cell approach. Finally, our code holds all tree data non-overlappingly and does not require any rank to hold the subdomain dimensions of each and every other rank. This makes the present algorithms differ from classic codes that rely on SFCs to obtain a fine grid partition and then construct (overlapping) coarse grid regions bottom-up. All adjacency data of the domain decomposition is localised.

Comparisions to several of these aforementioned approaches with tunneling highlight the present algorithms’ strenghts and weaknesses. Notably, they highlight that our local decomposition and particle handling is, for many applications, superior to classic SFC decompositions where the domain decomposition is known globally as well as to approaches where particles are handed around cyclically or sorted from a coarse regular mesh into the fine AMR structure [4]. A comparison to a classic linked-cell code that does not support tunneling finally reveals the price we have to pay to enable tunneling.

Throughout the present paper, we rely on our open source AMR software framework Peano [30, 31]. It allows us to rapidely implement and evaluate the present algorithmic ideas. raPIDT was fed back into this framework and now is available as black-box particle handling scheme. Yet, none of the present algorithms is tied to that particular framework. They work for any spactree-based code—relying on classic quadtree or octree discretisation, e.g.—and for any spatial dimension as long as two criteria are met. First, our algorithms need all resolution levels of the multiple AMR grids embedded into each other explicitly. The meshing software has to provide all the mesh resolution levels to the solver and the solver has to be able to embed data into each and every level. Providing solely the finest mesh, e.g., is not sufficient. Second, the algorithms have to switch from coarser to finer levels and the other way round throughout the mesh traversal. Notably, a depth-first, a breadth-first or any hybrid traversal on the tree that is implemented recursively work.

3. Use case: a particle-in-cell code

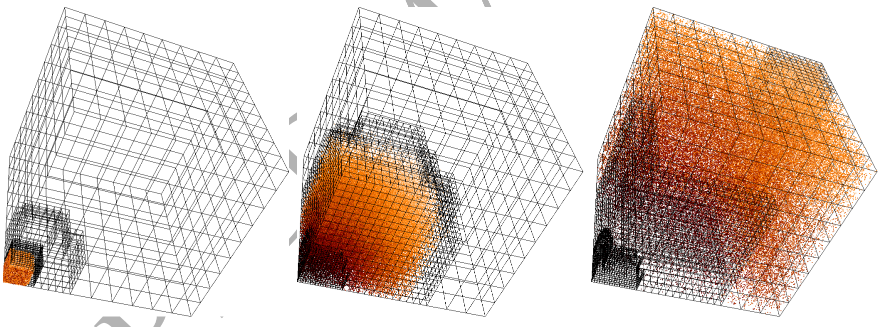


Figure 1: An artificial test setup at time steps $t \in \{2, 130, 190\}$ from left to right: Particles are homogeneously distributed among a small cube embedded into the unit cube at time $t = 0$. We apply reflecting boundary conditions. As the particles move according to a fixed random velocity (the brighter the higher their speed) and as the grid keeps the number of particles per cell bounded, the mapping of grid entities to particles is permanently to be updated.

The particle-in-cell (PIC) method has originally been developed to solve kinetic equations in plasma physics and self-gravitating systems (the so-called Vlasov equation, or collisionless Boltzmann equation), but it has also shown to be efficient for computational fluid simulations ([5, 6, 9, 10, 13, 14, 16, 19, 27], e.g.). Macro particles in a Lagrangian frame—they mimic the behaviour of a distribution function—are tracked in continuous phase space. Simultaneously, moments of the distribution function such as charge density for plasma physics simulations are computed on an Eulerian frame (fixed cells) in $\mathbb{R}^d$, $d \in \{2, 3\}$.

The PIC method reduces the N-body problem by filtering out binary interactions between particles through a so-called mean field approximation. In return, it couples the particles to a grid accommodating a grid-based solver of a partial differential equation. PIC typically reads as follows:

1. Given a set of particles at positions x_i, the charge density ρ is defined on the grid, and obtained from the particle through $\rho = \sum_i R(x_i)$. The restriction $R(x_i)$ in our case has local support and the grid based upon squares/cubes is chosen such that R affects at most 2^d vertices. It depends on the particle position.
2. We solve
$$\mathcal{L}(V) = \rho \tag{1}$$
on the grid which yields a potential V. V's semantics depend on the model. In our case it is the electric potential though all properties hold when we consider a gravitational potential with ρ being the *mass* density. $\mathcal{L}$ is, in our case, the Laplacian. Throughout the solution process, the grid is dynamically coarsened and refined to resolve V and the particle-cell interplay accurately.
3. A field $E = -\nabla V$ is derived from the potential. This field then is interpolated from the grid to the particle positions $\forall i : \quad E_i = R^T(x_i)E$. We apply the transpose of the interpolation per particle.
4. The particle velocities v_i and positions x_i are updated (some authors call this "push" [19]) with $\partial_t v_i = \phi_i(E_i, v_i)$ and $\partial_t x_i = v_i$ with a generic particle property equation $\phi_i(E_i, v_i)$.

Moving the particles is computationally cheap once the impact E_i on the particles is known. The particle-to-grid and grid-to-particle mappings usually stem from the multiplication of particle shape functions translated to the particle positions x_i with grid-aligned test functions that in turn are used to discretise the PDE. In our case, we restrict to Cartesian grids, i.e. to square or cube cells for the spatial discretisation, and we use d-linear shape functions for the PDE. More sophisticated schemes are possible.

As $\mathcal{L}$ generally may comprise $\partial_t E$, E or V from the previous time step have to be available on the grid. Equation (1) typically is solved by an iterative scheme. The present work neglects particularities of the PDE solver as well as dynamic load balancing and adaptivity criteria. However, we highlight that the impact of the particles on the PDE solution correlates to the particle density, i.e. the more particles in a given grid region, the rougher the PDE solution due to stimuli on the right-hand side. At the same time, a proper adaptivity criterion anticipates the Debye length: the maximum grid size depends locally on both the particle density and the mean velocity (bulk flow velocity) or mean square particle velocity (thermal velocity) [2]. Our particle-grid realisations have to support dynamically adaptive grids.

We first focus exclusively on adaptive Cartesian grids and storage paradigms for the dynamically adaptive grid holding the particles in the cells/vertices. Different to many other particle codes where the grid is merely a helper data structure, our grid stays persistent between any two time steps. Second, we focus on reassignment procedures once the particles are moved. Different to many other particle codes, it is crucial that the grid-particle relations are updated immediately and that there is no restriction on v_i with respect to the grid size. Third, we focus on the distributed memory parallelisation of the reassignment induced by a non-overlapping, given grid decomposition, and we study how particles have to be exchanged if they move along this decomposed geometric structure. This facilitates a fast evaluation of the grid-particle operator, as particles reside on the same node as their corresponding grid element. A discussion of a proper choice of a grid decomposition as well as dynamic load balancing are beyond scope. Finally, we reiterate that we neither can make assumptions on the grid structure nor on the particle velocities nor on the actual particle distribution. Particles can either traverse the grid elements smoothly or tunnel several grid elements a time.

4. A distributed spacetree data structure holding particles

A multitude of ways exists to formalise and implement adaptive Cartesian grids. Tree-based approaches are popular (cf. overview in [1] or [15, 20, 21, 29, 32]). They facilitate dynamic adaptivity and low memory footprint storage schemes teaming up with good memory access characteristics—in particular in combination with space-filling curves [1].

In the present paper, we follow a k-spacetree formalism [29, 32]: the computational domain is embedded into a square or cube, the *root*. This geometric primitive is split into k parts along each coordinate axis. We end up with k^d squares or cubes, respectively. They tessellate the original primitive. This setup can be represented by a graph with $k^d + 1$ nodes and a relation $\sqsubseteq_{child\ of}$. Each node of this graph represents one cube or square, respectively, and is denoted as *cell*. If $a \sqsubseteq_{child\ of} b$, a is contained within b and is derived from b by k cuts through b along each coordinate axis. We continue recursively while we decide for each cell whether to refine further or not. The resulting graph is a k-spacetree given by $\sqsubseteq_{child\ of}$, a set of cells $\mathcal{T}$ and a distinguished root. For $k = 2$ the scheme mirrors the traditional octree/quadtree concept [1, 15, 20, 21]. Our present code relies on the software Peano [30, 31] and thus uses $k = 3$. However, all algorithmic ideas work for any $k \geq 2$.

The distance from the root cell to any other cell is the cell's *level*. All cells of one level represent geometric primitives of exactly the same size, are aligned and do not overlap. The k-spacetree consequently yields a cascade of ragged Cartesian grids. The union of all these grids is an adaptive Cartesian grid. Let Ω_ℓ denote the grid of one spacetree level ℓ. The union $\Omega_h = \bigcup_\ell \Omega_\ell$ then yields an adaptive Cartesian grid. A cell of $\mathcal{T}$ is a *leaf* if it does not have a child. If $a \sqsubseteq_{child\ of} b$, b is a *parent* of a. Due to the cascade-of-grids formalism, there may be multiple vertices in a spacetree at the same spatial location while they belong to different grids Ω_ℓ, i.e. different levels. Each vertex has up to 2^d adjacent cells on the same level. If it has less than 2^d adjacent cells, it is a *hanging vertex*. Anticipating the spacetree's partial order, a vertex has up to 2^d parent vertices (Figure 2): A vertex v_b is a parent to vertex v_a, if all cells

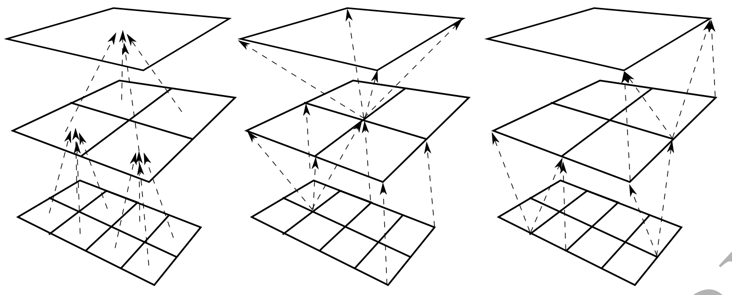


Figure 2: The spacetree introduces a parent relation on the spacetree cells (left; arrows point from child to parent). This spatial order in combination with the grid embedding induce a parent relation upon the vertices as well (middle and right). Vertices coinciding with coarser vertices on the same level have one parent, vertices within coarser cells have 2^d parents, vertices located on coarse grid faces and edges have a parent cardinality in-between.

that are adjacent to v_a have parent cells that are in turn adjacent to v_b. While many tree-based codes deduce an adaptive grid from a spacetree formalism and then work basically on the spacetree's leaves, we preserve and maintain the whole tree as computational data structure though the particle-grid interaction is often computed only in leaves.

4.1. Dual spacetree grid

Besides the cascade of Cartesian grids, the spacetree formalism also induces dual grids. A dual grid $\Omega_\ell^{(d)}$ of one level is defined as follows: It is a grid consisting of geometric primitives of exactly the type as in Ω_ℓ. They are dilated such that each cell center of Ω_ℓ coincides with one non-hanging vertex of the dual grid. $\mathcal{T}^{(d)}$ then is the cascade of dual grids to $\mathcal{T}$ (Figure 3). For odd k—we have $k = 3$—we observe *dual grid consistency*: A dual cell of one level either is contained completely within a dual cell of a coarser level or does not intersect with coarsers cells at all. The present algorithms did, in principle, not rely on this consistency, but their implementation's simplicity benefits from $k = 3$.

4.2. Storing particles within the spacetree

The present paper studies two choices to store the particle-grid relations: either each cell holds the particles covered by it, or each vertex holds the particles whose positions are closer to this vertex than to any other vertex. "Holds" denotes that the grid entity basically links to an array of particles. While we focus on the data handling and parallelisation, packed memory algorithms [5], e.g., can replace these arrays with more efficient realisation variants avoiding frequent reallocation. Furthermore, we do not discuss global data layout optimisations to improve the placement of the arrays in memory [5, 10] but rather refer to [29, 32] where we introduce a multiscale ordering of the spacetree cells along the Peano space-filling curve. Such an ordering of the cells transfers to an ordering of arrays. We thus can assume reasonably efficient memory data access with respect to caches. Without loss of generality, these arrays are, for

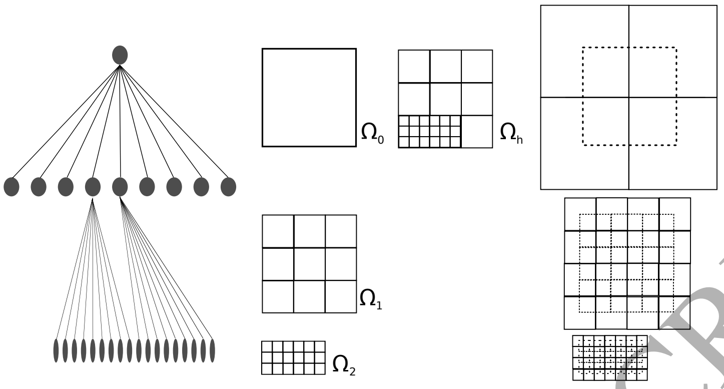


Figure 3: From left to right: A spacetree is a directed graph $\sqsubseteq_{child\ of}$ visualised bottom-up here. It yields a cascade of ragged Cartesian grids $\Omega_0, \Omega_1, \Omega_2$. The merger of the three is an adaptive Cartesian grid Ω_h. The dual $k = 3$-spacetree yields a cascade of three dual grids (with the original grid as dotted lines).

the time being, empty on the coarse levels, whereas all particles reside within the finest grid resolution. In our second storage scheme, particles are assigned to the vertex whose dual cell covers their position. It is a Voronoy-based particle assignment [24]. As we are working in a spacetree environment, the two approaches are named *particle in tree* (PIT) and *particle in dual tree* (PIDT).

4.3. Parallel grid decomposition

Let $\mathbb{P} = \{p_0, p_1, \ldots\}$ be the finite set of processes (ranks) on a parallel computer. $col : \mathcal{T} \mapsto \mathbb{P}$ is a *colouring* that assigns each spacetree cell a colour, i.e. a processor that is responsible for this cell. In practice, each processor holds only that part of a spacetree it is responsible for [29]. Let $\sqsubseteq_{worker\ of}$ induce a tree topology on $\mathbb{P}$ as follows:

$$\forall a \sqsubseteq_{child\ of} b : \ col(a) = p_i \wedge col(b) = p_j \Rightarrow p_i = p_j \vee p_i \sqsubseteq_{worker\ of} p_j. \qquad (2)$$

As the graph in (2) shall be free of cycles, different subtrees of the global spacetree are handled by different processes, i.e. on different ranks. We decompose the spacetree. No refined cell is shared among multiple ranks, and *col* introduces a master-worker relations with a distinguished global master being responsible for the root. This scheme differs from local essential tree constructions where coarse grid cells are replicated among multiple nodes ([15] and their references to original work). We rely on a unique rank responsibility for coarse grid spacetree cells [30].

Our tree colouring induces a multiscale non-overlapping spacetree decomposition, i.e. each cell is assigned to a rank uniquely, while vertices at the partition boundaries are held and replicated on up to 2^d ranks. The term non-overlapping refers to each individual level. Let each cell of $\mathcal{T}^{(d)}$ be assigned to the ranks that hold cells that are adjacent to the center of the cell in $\mathcal{T}$. The colouring then induces a multiscale 1-overlapping decomposition on $\mathcal{T}^{(d)}$.

For PIC, it is convenient to decompose particles along *col* as the particles do not interact. For PIDT, dual cells intersecting with the parallel boundary then are replicated while the particles are never replicated but always are assigned to one rank uniquely.

4.4. Event-based formalism of the tree traversal

A *tree traversal* is an algorithm running through $\mathcal{T}$. An effective tree traversal shall have three properties:

1. All data access is local within the grid/tree.
2. All data access anticipates the domain decomposition.
3. The data access scheme is efficient. In particular, any particle sorting shall be possible in one tree traversal even if the grid changes.

For this, we process each cell of this set twice: an operation `enterCell` is performed prior to an operation `leaveCell`. We enforce

$$\begin{array}{rll} \forall a \sqsubseteq_{child\ of}\ b: & \Rightarrow \quad \texttt{enterCell}(b) \sqsubseteq_{before} \texttt{enterCell}(a)\ \wedge & \\ & \quad\ \ \texttt{leaveCell}(a) \sqsubseteq_{before} \texttt{leaveCell}(b) & \text{and} \\ \forall c: & \quad\ \ \texttt{enterCell}(c) \sqsubseteq_{before} \texttt{leaveCell}(c) & \text{for } a,b,c \in \mathcal{T} \end{array} \quad (3)$$

where $\sqsubseteq_{before}$ is a partial temporal access order. Obviously, both depth-first and breadth-first k-spacetree traversal as well as hybrid variants preserve (3).

As the tree is a representation of the cascade of grids, a tree traversal describes a strict element-wise multiscale processing of the whole grid cascade. For any element-wise realisation of a PDE solver or a particle-based algorithm it is then sufficient to specify which data are assigned to the k-spacetree's vertices and cells, and to specify how individual *events* [30] such as `enterCell` and `leaveCell` map onto algorithmic fragments often called compute kernels. Our discussion restricts to element-wise algorithms as those algorithms fit straightforwardly to non-overlapping domain decompositions.

Following the notion of element-wise processing, only records assigned to one cell and its adjacent vertices are available to an event. Following the notion of a spacetree, the parent data are passed to events as well. Besides the cell events, our implementations rely on two additional events: `touchVertexFirstTime` is called once per spacetree vertex per traversal per rank before the vertex is used by this rank the very first time, i.e. before `enterCell` is invoked on any adjacent cell of this vertex. `touchVertexLastTime` is called once per spacetree vertex on each rank after the vertex has been used the very last time, i.e. after `leaveCell` has been invoked on all adjacent cells.

In a parallel tree traversal, the global master starts to traverse the tree. Its workers' tree traversals are successively started up as soon as (3) allows for. This is a broadcast along a tree topology. Whenever a tree traversal ascends again in the tree (processes `leaveCell` and `touchVertexLastTime`), it might have to wait for other colours to finish their share of the global tree due to (3). The startup of a remote colour is accompanied by the two events `prepareSendToWorker` and `mergeWithWorker` invoked on the master or worker, respectively. The other way round, `prepareSendToMaster` and `mergeWithMaster` are called. Furthermore, events, i.e. plug-in points for the algorithm, do exist to merge vertices at the domain decomposition boundaries. Vertex exchanges are triggered after a local vertex has been used for the last time. `mergeWithNeighbour` events then are invoked on each rank per parallel domain boundary vertex prior to any usage of this vertex. As `mergeWithNeighbour` is called prior to the next usage but copies are sent after `touchVertexLastTime`, vertex data exchange along the rank boundaries overlaps two iterations. This exchange is asynchronous while the information exchange between masters and workers is synchronous.

5. Cell- and vertex-based particle movers

For each of the two particle storage schemes, PIC requires the scheme to maintain the particle-to-grid mapping. Whenever a particle moves, our code has to analyse whether the particle remains within its cell or dual cell, respectively, or it has to update the mapping otherwise. While it might be straightforward to iterate first over all particles to move them before we sort them in an update sweep, we propose to merge particle movement and reassignment. This way, we avoid an extra sorting step. For both PIT and PIDT, we propose to move particles up and down within the spacetree to enable tunneling, but we enforce that all particles are sorted into the leaf tree level prior to any subsequent operation on the particle in the next traversal. Our algorithmic sketches describe stationary grids. If spacetree nodes are added dynamically, both approaches automatically move particles into the right grid entities as the sort algorithm makes all particles reside on the finest grid level. If spacetree nodes are removed, both approaches move the particles associated with removed grid entities up in the spacetree and continue. Support of dynamic adaptivity hence is straightforward and not discussed further.

5.1. Particle in tree (PIT)

PIT maps each particle from the particle set $\mathbb{M}$ onto a leaf of the spacetree, i.e. $\mathcal{X}_{\text{PIT}} : \mathbb{M} \mapsto \mathcal{T}$. Only the particle position $x \in \mathbb{R}^d$ determines this mapping. PIT's particle update then reads as Algorithm 1 and integrates into any tree traversal preserving (3) (Figure 4).

Each global particle sorting is split among two traversals. Let traversal t_2 follow t_1. All lift operations with respect to particle updates in t_1 are embedded into the traversal t_1, whereas the drops and the particle position updates are embedded into t_2. t_2 already realises the lifts of the subsequent time step whereas t_1 realises the drops of the previous one. Hence, one (amortised) traversal per resort is sufficient—the first 1.5 traversals realise the first sort, the next 2.5 the second, ...—and the following statement holds:

Theorem 5.1. *Whenever* `enterCell` *is invoked on a leaf a and all its preamble operations terminate, all particles within the cell have a position x covered by a.*

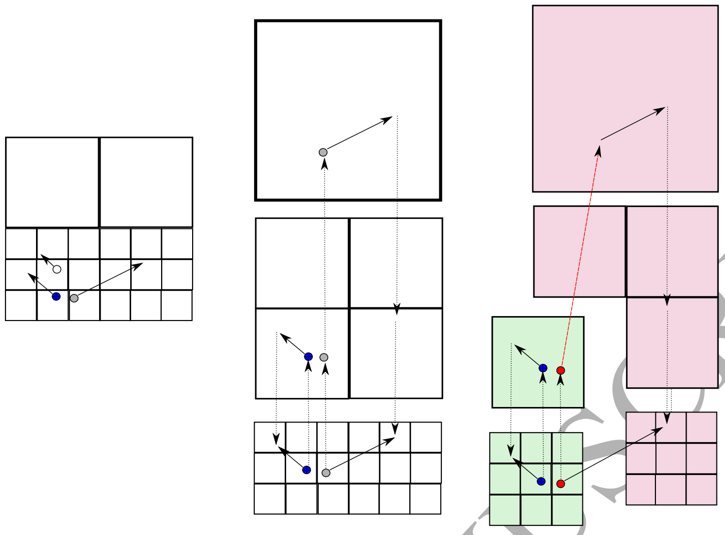


Figure 4: Left: Assigning particles to cells is trivial as long as particles due not leave 'their' cell to the particle update (empty particle). If particles leave a cell (filled particles), they have to be reassigned to a new cell. Middle: PIT lifts these particles to the next coarser levels and then drops them back. If particles tunnel (light gray), they have to be lifted several levels. Right: A tree decomposed into two colours is traversed by PIT. Lifts (red) and drops (blue) are embedded into this parallel traversal. Therefore, particles are exchanged in-between processors only along the parallelisation's master-worker topology.

Proof. We restrict to a single particle $m \in \mathbb{M}$, and we assume that the theorem holds prior to traversal t_1 with $\mathcal{X}_{\text{PIT}}(m)$ being a leaf. We first show that $x \in \mathcal{X}_{\text{PIT}}(m)$ as soon as t_1 terminates though the image can be a refined spacetree cell. $x \in \mathcal{X}_{\text{PIT}}(m)$ denotes that the spacetree cell to which m is mapped to covers the position x. For this, we make a simple top down induction on the tree depth. If $\mathcal{X}_{\text{PIT}}(m)$ is the root, $x \in \mathcal{X}_{\text{PIT}}(m)$ is trivial as the root spans the whole computational domain. Otherwise we distinguish two cases for `leaveCell` invoked for a spacetree cell of level $\ell + 1$:

- $x \in \mathcal{X}_{\text{PIT}}(m)$: the particle resided in $\mathcal{X}_{\text{PIT}}(m)$ prior to a position update and doesn't leave the cell. PIT does not modify $\mathcal{X}_{\text{PIT}}$.

- $x \notin \mathcal{X}_{\text{PIT}}(m)$: the particle leaves the cell in which it was contained before. In `leaveCell`, we assign it to its parent, i.e.

$$\begin{array}{rcll} \mathcal{X}_{\text{PIT}} & \mapsto & \mathcal{X}^{(\text{new})}_{\text{PIT}} & \text{with} \\ \mathcal{X}^{(\text{new})}_{\text{PIT}}(m) & = & p & \text{while} \\ \mathcal{X}_{\text{PIT}}(m) & \sqsubseteq_{child\ of} & p. & \end{array}$$

  Due to (3), this happens after `enterCell` is invoked for m in t_1 and does not harm our initial assumption. Due to (3), this transition is triggered before the `leaveCell` call for the parent on level ℓ. The statement for level ℓ holds by induction.

Algorithm 1 Particle-in-tree algorithm.

```
function enterCell(cell $c \in \mathcal{T}$)
    if $c$ refined then
        for all $m \in \mathbb{M}$ associated to $c$ do          ▷ preamble
            identify $c' \sqsubseteq_{child\ of}$ $c$ containing $m$
            assign $m$ to $c'$                                   ▷ drop
        end for
    end if
    invoke application-specific operations
    if $c$ unrefined then
        for all $m \in \mathbb{M}$ associated to $c$ do
            update position $x(m)$                               ▷ move
        end for
    end if
    invoke application-specific operations
end function
function leaveCell(cell $c \in \mathcal{T}$)
    invoke application-specific operations
    for all $m \in \mathbb{M}$ associated to $c$ do              ▷ epilogue
        if position $x$ of $m$ not contained in $c$ then
            assign $m$ to $c'$ with $c \sqsubseteq_{child\ of}$ $c'$   ▷ lift
        end if
    end for
end function
```

For the subsequent traversal t_2, we again use an induction over the tree height and distinct two cases. However, we argue bottom-up. For trees of height zero the algorithm's correctness is trivial. Let the root be refined.

- $\mathcal{X}_{\text{PIT}}(m)$ on level ℓ is a leaf: The theorem holds.
- $\mathcal{X}_{\text{PIT}}(m)$ on level ℓ is refined. The algorithm bucket sorts the particle into the child cell covering its new position, i.e.

$$\begin{array}{rcll} \mathcal{X}_{\text{PIT}} & \mapsto & \mathcal{X}_{\text{PIT}}^{(\text{new})} & \text{with} \\ \mathcal{X}_{\text{PIT}}^{(\text{new})}(m) & = & c & \text{while} \\ c & \sqsubseteq_{child\ of} & \mathcal{X}_{\text{PIT}}(m). & \end{array}$$

  c has level $\ell + 1$. Due to (3), `enterCell` for the parent is invoked prior to `enterCell` for the spacetree node on level $\ell + 1$. The statement for level $\ell + 1$ holds by induction.

□

A distributed memory parallel version of PIT adds two case distinctions. Whenever PIT lifts a particle from a local root, i.e. a spacetree node whose parent is assigned to a different colour, this particle is sent to the parent's rank by the event `prepareSendToMaster`. In return, `mergeWithMaster` receives and inserts it into the local data structure. Whenever the tree traversal encounters a cell of a different colour, `prepareSendToWorker` replaces the drop by a send to the worker rank. `mergeWithWorker` receives them and continues to drop them (Figure 4). All

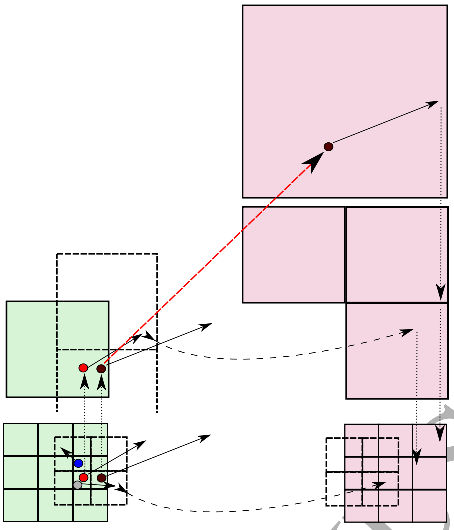


Figure 5: Parallel PIDT: There are three different types of particle movements. Particles might remain in their dual cell (blue), might move to an adjacent dual cell (gray) or tunnel (red and dark red). Only for the latter, multilevel movement comes into play. Copies of particles leaving through vertices without further tunneling are sent away immediately throughout the traversal but are not received prior to the next iteration (asynchronous communication, dotted arrows from left to right). Only very few tunneling particles are communicated synchronously among the tree topology (dark red particle, red dotted diagonal arrow).

parallel data flow is aligned with the tree topology on $\mathbb{P}$, i.e. particles are only sent up and down within the spacetree. All data exchange is synchronous.

5.2. Particle in dual tree (PIDT)

PIDT augments each particle by a boolean flag *moved* $\in \{\top, \bot\}$ ("has moved already" $\top$ and "has not not moved yet" $\bot$) and maps each particle onto a vertex of the spacetree, i.e. $\mathcal{X}_{\text{PIDT}} : \mathbb{M} \mapsto \mathcal{T}^{(d)}$. The particle update then reads as as Algorithm 2 and integrates into any tree traversal preserving (3) (Figure 5). For efficiency reasons, the particle loops in `enterCell` and `leaveCell` can be merged for leaf cells.

Theorem 5.2. *Whenever* `touchVertexFirstTime` *is invoked on a vertex v and all its preamble operations have terminated, all particles within the dual cell of v have a position covered by this dual cell.*

Proof. The proof follows the proof of Theorem 5.1. □

Again, parallel PIDT is straightforward. Whenever the grid traversal has invoked `touchVertexLastTime` on one parallel rank for a vertex, each particle of this vertex falls into one of four categories:

- It is to be lifted but the vertex does not belong to the coarsest level on the local rank. These particles are neglected by the parallelisation as the

Algorithm 2 Particle-in-dual-tree algorithm (continued in Algorithm 3).

```
function touchVertexFirstTime($v$)
    if $v$ refined then                                   ▷ i.e. all surrounding cells are refined
        for all $m \in \mathbb{M}$ associated to $v$ do                      ▷ preamble
            $moved(m) \leftarrow \perp$
        end for
    end if
    invoke application-specific operations
end function
function enterCell(cell $c \in \mathcal{T}$)
    if $c$ refined then                                                      ▷ preamble
        for all $2^d$ adjacent vertices $v$ do
            for all $m \in \mathbb{M}$ associated to $v$ do
                if $moved(m) = \perp$ and $x(m)$ contained in $c$ then
                    identify child vertex $v'$ whose dual cell holds $x(m)$
                    assign $m$ to $v'$                                       ▷ drop
                end if
            end for
        end for
    end if
    invoke application-specific operations
    if $c$ unrefined then
        for all $2^d$ adjacent vertices $v$ do
            for all $m \in \mathbb{M}$ associated to $v$ do
                if $moved(m) = \perp \wedge x(m)$ contained in $c$ then
                    update position $x$ of $m$                               ▷ move
                    $moved(m) \leftarrow \top$
                end if
            end for
        end for
    end if
    invoke application-specific operations
end function
```

particle is lifted locally with `touchVertexLastTime` being called bottom-up throughout the traversal.

- It is to be lifted and the vertex is adjacent to the coarsest cell held by a rank. Such a particle is sent to the rank's master node where it is received in the same iteration prior to the master's `leaveCell`.

- It belongs to the vertex's dual cell and intersects the local domain. Such a particle already is assigned to the correct vertex and skipped.

- It belongs to the vertex's dual cell but it is not covered by the local domain. In the latter case, the particle is removed from the local vertex and sent to the respective destination rank merging all received particles into the local vertices prior to `touchVertexFirstTime` in the next tree traversal.

The data flow from workers to masters is aligned with the tree topology on $\mathbb{P}$. This data flow comprises only particles that have to be lifted between the coarsest worker cell and its parent residing on the master. It is synchronised with the tree traversal. The drop mechanism also uses synchronous particle exchange

Algorithm 3 Particle-in-dual-tree algorithm continued from Algorithm 2.

```
function leaveCell(cell $c \in \mathcal{T}$)
    invoke application-specific operations
    for all $2^d$ adjacent vertices $v$ do                                   ▷ epilogue
        for all $m \in \mathbb{M}$ associated to $v$ do
            if $moved(m) = \top \wedge x(m)$ contained in dual cell of other $v'$ adjacent to $c$ then
                assign $m$ to $v'$                              ▷ linked-list type reassignment
                                                                           ▷ (no tunneling)
            end if
        end for
    end for
end function
function touchVertexLastTime($v$)
    invoke application-specific operations
    for all $m \in \mathbb{M}$ associated to $v$ do                           ▷ epilogue
        if $moved(m) = \top \wedge x(m)$ not contained within dual cell of $v$ then
            assign $m$ to parent vertex                            ▷ lift (cmp. Figure 2)
                                                                              ▷ tunneling
        end if
    end for
end function
```

whenever a rank descends into a cell not handled locally. The exchange through the vertices follows a Jacobi-style update: particles are sent away in one traversal and received prior to `touchVertexFirstTime` of the subsequent traversal. That is convenient, as the particle lift and drop are split among two tree traversals. The latter data exchange is asynchronous and can be realised in the background in parallel to the tree traversal.

5.3. Remarks

While PIT and PIDT realise the same fundamental ideas, they differ in code complexity and the handling of not-tunneling particles. If particles do not tunnel but move from one cell to a neighbouring cell, PIT moves them up in the tree one level and down one level. PIDT exchanges them directly mirroring linked-list techniques. This pattern applies in a multiscale sense for PIDT. PIDT thus is a multiscale linked-list approach, where its "radius of operation" is twice the grid width. It is expected to have fewer lifts and drops compared to PIT. Contrary, PIDT's higher algorithmic complexity mirrors directly to a more complex code. While this might not influence the runtime, the redundant check of particles does: we need a flag *moved* to mark updated particles which might double the number of particle loads, i.e. memory accesses.

Finally, we antedate experimental insight. Both algorithms suffer from a synchronisation of masters and workers. Workers cannot start their tree traversal prior to their master having entered the parent cell, as particles might drop from coarser resolution levels. Masters cannot continue their tree traversal before the workers have finished, as particles might have to be lifted from finer resolution levels. Both synchronisation points introduce a latency and a bandwidth penalty and make perfectly balanced load a must. The bandwidth penalty scales with the number of particles to be exchanged. It is hence less significant for PIDT than for PIT. Both master-worker and worker-master communication describe a global all-to-all communication that is realised as tree communication. We state that the resulting global synchronisation were needless if no particles would be lifted or dropped.

5.4. Restriction-avoiding PIDT (raPIDT)

To identify cases where we could skip the global communication, we introduce a marker $v_{max} : \mathcal{T} \mapsto \mathbb{R}_0^+$. On any leaf of the spacetree, v_{max} shall hold the maximum velocity component of all particles associated to the 2^d adjacent vertices. v_{max} is a cell-based value though it results from vertex-associated data. It is the maximum norm of all velocity components of all particles contained within the 2^d dual cells intersecting the current cell. This value is determined by `leaveCell` in each traversal. On a refined cell, we update the value with

$$v_{max}(c) \leftarrow max\{v_{max}(c') : c' \sqsubseteq_{child\ of}\ c\}. \qquad (4)$$

Here, $v_{max}(c)$ is an analysed tree attribute [11] (re-)computable on-the-fly.

Given any spacetree cell of width $h = 3^{-level}$, the time step size Δt and a correct v_{max}, we know that no particles will be removed from this cell or any of its children and successors due to lifts if all particles contained have a velocity smaller than $h/\Delta t$. We can check this due to v_{max}. We augment PIDT with the following mechanisms:

- Each rank holds a boolean map for all its workers.
- On `enterCell` for a cell deployed to a worker, we first update the local v_{max} due to all particles dropped into this cell.
- The updated $v_{max} \leq h/\Delta h$ identifies a priori whether particles will be lifted again from the remote subtree. We store this information within the local boolean map.
- We then continue with PIDT, i.e. start up the remote rank.
- On `leaveCell` on the respective deployed cell, we have two opportunities:
  1. If lifts are to be expected, we receive all particles from the worker, redetermine (4) and continue.
  2. If no lifts are to be expected, we continue with the tree traversal without waiting for any worker data. The reduction is avoided.

The receive branch is mirrored with the same rules on the worker to ensure that no worker sends data in the reduction-avoiding case at all.

6. Results

Our case studies were executed on SuperMUC at the Leibniz Supercomputing Centre and the N8 Polaris system at Leeds. SuperMUC hosts Sandy Bridge E5-2680 processors clocked to 2.3 GHz, Polaris hosts Sandy Bridge E5-2670 processors at 2.6 GHz plus Turbo Boost. Both have two eight-core processors per node. SuperMUC runs Intel MPI 4.1, and every 8192 cores (512 two-processor nodes) are called an island and are connected via a fully non-blocking Infiniband network. Beyond that core count, SuperMUC relies on a 4:1 blocking network. Polaris runs Open MPI 1.6.1, and every 192 cores are fully non-blocking connected to one Mellanox QDR InfiniBand switch. Beyond that core count, Polaris realises 2:1 blocking. Our code relies on good, i.e. close to optimal, placement, i.e. it uses as few switches as possible. All performance data is specified as particle updates per second, all machine sizes are specified in cores.

For the performance tests, we remove the PDE solver part as well as the interplay of particles and grid, and thus focus exclusively on the particle handling. The appendix presents a real-world run. We study worst-case setups where the particle handling's performance characteristics are not interwoven with other application phases. Yet, we artificially impose, where highlighted, a fixed number of 0, 128, 256, 1024 or 4096 floating point operations (flops) onto each particle move to quantify the impact of arithmetics on the runtime. All particles are initially placed randomly and homogeneously within the unit square $(0,1)^d \subset \mathbf{R}^d$ or a subdomain $(0.1, 0.1)^d \subset (0,1)^d$. The total number of particles results from the particle density $\rho_{\text{particles}}$ within the initially populated area. Each particle is assigned a random velocity $0 \leq |v| \leq 1$ uniformly distributed (Figure 1). This way, we make the characteristic particle movement per step depend only on one quantity—the time step size. A confusion with particle-grid mapping in the PIC blueprint of Section 3 is out of question as we neglect the PDE. We hence write $\rho = \rho_{\text{particles}}$. Time step size Δt and the maximum number of particles per cell (`ppc`) both determine the particle movement/tunneling and the grid structure. Whenever the given `ppc` is overrun in a particular spacetree cell, this cell is refined and the particles are hence sorted into the new spacetree nodes. This mirrors dynamic adaptivity assuming that the particle density correlates to the smoothness of E in (1). Whenever multiple children of one refined spacetree node can be coarsened without violating the `ppc` constraint, we remove these children and lift the particles into the formerly refined cell. All experimental code supports dynamic AMR. All experiments apply reflecting boundary conditions at the border of the unit square or cube, respectively, but do not change the particle velocities otherwise. All experimental data result from a code with a static domain decomposition deriving $\mathbb{P}$ from the spacetree due to graph partitioning. The experiments are, if not stated otherwise, stopped after few time steps and thus do not suffer (significantly) from ill-balancing. All domain decomposition/initialisation overhead is removed from the measurements.

With all these parameters at hand, we can study the impact of total particle count relative to the computational domain, we can study setups with extremely inhomogeneous particle distribution vs. homogeneous particle distributions, we can analyse the impact of the ratio of particle speed to minimal grid size, and we can study the interplay of particle density, ppc and time step size.

6.1. *Algorithmic properties*

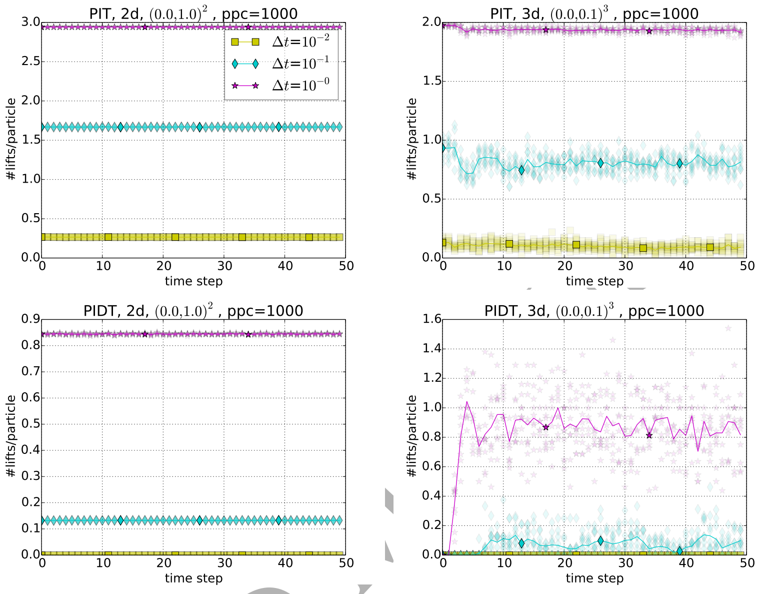


Figure 6: Lifts per particle per time step (ppc=1000). Left column: particles are homogeneously distributed among the unit square ($d = 2$). Right column: particles are initially homogeneously distributed in $(0, 0.1)^3 \subset \mathbb{R}^3$, i.e. the grid is dynamically adaptive. Results from ten random initial setups (blurry) with averages as solid lines. No lifts are observed at all for $\Delta t \leq 10^{-3}$.

Prior to measuring the runtime, we focus on the lift behaviour for different setups and count the average number of lifts per particle during the first 50 time steps. Drop observations deduce from these. All measurements are almost independent of (sufficiently big) ρ—here set to 10^7. If a particle is lifted n levels up in the spacetree, we count this as n independent lifts. When we fix ppc (ppc=1000 in Figure 6, e.g.), we observe that the number of lifts remains de facto invariant throughout the simulation for a globally homogeneous particle distribution. For an inhomogeneous initial distribution, lifts occur only sporadically (unless the time step size is very large), i.e. the average number of lifts is below one. For both realisation variants, the number of lifts scales with the time step size, i.e. the bigger the time step size the more lifts per particle, while PIT yields significantly more lifts than PIDT. If the time step sizes underrun a certain threshold, PIDT comes along completely without lifts while PIT has a very low lift count.

If we fix in turn the time step size and make ppc a free parameter (see Appendix B), we observe that any ppc yields, on average, time-independent behaviour for homogeneous setups. The bigger ppc the fewer lifts, and the number of lifts per particle is always bigger for PIT than for PIDT where the number of lifts is negligible for reasonably big ppc or coarse grids, respectively. Larger ppc make each leaf represent larger geometric domains and thus explain the ppc dependence. If the particle distribution is inhomogeneous, the corresponding grid at setup time is strongly adaptive if ppc is reasonably small. Initial time steps of PIDT then face almost no lifts; similar as in the homogeneous setup. However, the number of lifts increases as the grid becomes more regular (and coarser), and the particles distribute more homogeneously, until the lift count drops again to its regular characteristics. With increasing ppc the curve flattens out and shifts to the right.

Due to the invariant velocity profile besides the reflecting boundary conditions, particles are expelled from the subarea by the large (electric) field. Therefore, the global grid determined by ppc becomes coarser and more regular (Figure 1). It smoothens out. As a consequence, subtrees of a certain depth hold more and more particles with velocities of the same magnitude. In return, the average particle density within each cell decreases. The further (fast) particles move away from the initially dense area, the more often they hit adaptivity boundaries, i.e. hanging nodes. PIDT has to lift them there. For PIT, these additional lifts make almost no difference compared to the lifts required anyway. Particles hit adaptivity boundaries often if ppc is small. As the grid smoothens out, also the lift counts drop.

Referring to real-world runs (Appendix A), such a behaviour mirrors a setup where initially all particles are held within the subarea where the solution to (1) depending on the particle density is non-smooth and yields large particle accelerations. The push out from this area then results from a mixed neutralising positively charged background and negatively charged particles.

6.2. Memory throughput

Table 1: Stream particle throughput, i.e. particles per second, for different particle counts p using 1,2,4,12,16 cores or 16 cores plus hyperthreading (32) on SuperMUC. The upper section gives results for $d = 2$, the lower for $d = 3$. Best case throughputs per row are bold.

p	1	2	4	8	12	16	32
10^4	$\mathbf{3.03 \cdot 10^7}$	$2.98 \cdot 10^7$	$4.14 \cdot 10^6$	$2.46 \cdot 10^7$	$1.46 \cdot 10^7$	$2.21 \cdot 10^7$	$1.94 \cdot 10^7$
10^5	$6.66 \cdot 10^7$	$1.03 \cdot 10^8$	$\mathbf{1.35 \cdot 10^8}$	$1.06 \cdot 10^8$	$8.96 \cdot 10^7$	$1.02 \cdot 10^8$	$6.15 \cdot 10^7$
10^6	$7.03 \cdot 10^7$	$1.16 \cdot 10^8$	$1.58 \cdot 10^8$	$1.56 \cdot 10^8$	$\mathbf{2.25 \cdot 10^8}$	$2.41 \cdot 10^8$	$5.51 \cdot 10^7$
10^7	$7.14 \cdot 10^7$	$1.25 \cdot 10^8$	$1.63 \cdot 10^8$	$2.44 \cdot 10^8$	$\mathbf{2.63 \cdot 10^8}$	$2.42 \cdot 10^8$	$7.99 \cdot 10^7$
10^8	$7.17 \cdot 10^7$	$1.26 \cdot 10^8$	$1.68 \cdot 10^8$	$2.28 \cdot 10^8$	$2.65 \cdot 10^8$	$\mathbf{2.84 \cdot 10^8}$	$2.59 \cdot 10^8$
10^4	$2.48 \cdot 10^7$	$3.22 \cdot 10^7$	$\mathbf{3.25 \cdot 10^7}$	$2.39 \cdot 10^7$	$2.42 \cdot 10^7$	$2.27 \cdot 10^7$	$1.44 \cdot 10^7$
10^5	$5.11 \cdot 10^7$	$8.43 \cdot 10^7$	$1.14 \cdot 10^8$	$\mathbf{1.23 \cdot 10^8}$	$7.67 \cdot 10^7$	$2.44 \cdot 10^7$	$2.43 \cdot 10^7$
10^6	$4.93 \cdot 10^7$	$9.15 \cdot 10^7$	$1.18 \cdot 10^8$	$1.73 \cdot 10^8$	$1.81 \cdot 10^8$	$\mathbf{1.94 \cdot 10^8}$	$4.56 \cdot 10^7$
10^7	$5.40 \cdot 10^7$	$9.30 \cdot 10^7$	$1.24 \cdot 10^8$	$\mathbf{1.82 \cdot 10^8}$	$1.77 \cdot 10^8$	$1.63 \cdot 10^8$	$9.24 \cdot 10^7$
10^8	$5.41 \cdot 10^7$	$9.33 \cdot 10^7$	$1.25 \cdot 10^8$	$1.65 \cdot 10^8$	$1.84 \cdot 10^8$	$\mathbf{1.92 \cdot 10^8}$	$1.65 \cdot 10^8$

To be able to put runtime measurements into context, we first run a benchmark holding an array of particles that is iterated once per time step without any grid. Each particle position is updated according to an explicit Euler integration step, it is reflected at the domain boundaries, and the resulting position is written back to the corresponding array position. Besides the position update, no computation or assignment to grid entities is done and no data is reordered. The implementation is the same source code fragment we use in the spacetree algorithms. As we aim to compare it with our parallel implementation running multiple MPI ranks per node, we parallelise this embarrassingly parallel benchmark with a plain parallel-for along the lines of the Stream benchmark [18]. This yields upper bounds, as our spacetree implementation relies on MPI only and thus has message passing overhead.

We analyse the throughput behaviour at hands of SuperMUC (Table 1). Polaris exhibits qualitatively the same behaviour. On both systems, a single core cannot exploit the memory subsystem alone. The bigger the particle count the more cores can be used effectively and the higher the throughput. However, the throughput does not scale linearly with the core count due to bandwidth restrictions. $3.00 \cdot 10^8$ for $d = 2$ and $2.00 \cdot 10^8$ for $d = 3$ are upper bounds for the throughput of our particle codes in the absence of any solver, i.e. of any 'real' computation, on SuperMUC. On Polaris, these best case thresholds have to be doubled due to the higher clock rate and the Turbo Boost. Besides for small particle numbers, less than eight threads/ranks per node do not make the nodes run into bandwidth saturation. As a consequence, all parallel experiments deploy six MPI ranks per node from hereon. This heuristic choice valid for both SuperMUC and Polaris yields a reasonable core usage while memory subsystem effects are not dominant. It also anticipates that the presence of a grid to maintain increases the average per-particle memory footprint.

6.3. Single core results

We next measure the particle throughput for homogeneous and inhomogeneous start scenarios with different Δt on a single core with 10^7 particles for 50 time steps, while we still neglect arithmetics per particle (Figure 7). PIT's throughput monotonously increases with increasing reasonable `ppc`. It decreases with increasing time step size. Any reduction of lifts due to a bigger `ppc` or smaller time step sizes pays off for a homogeneous start setup. For an inhomogeneous start setup, we observe a decreasing performance with bigger time step sizes as the grid then changes faster. This picture would change if we fixed the simulation time rather than the time step count. PIDT is typically outperformed by PIT due to the more complicated algorithm despite in situations where particles move very fast in an adaptive setting. `ppc` ≈ 1000 here yields the best throughput. Polaris and SuperMUC results do coincide if scaled with the clock rate, i.e. the duplication of throughput rates on Polaris cannot be observed again.

Hardware counter measurements with Likwid [26] reveal that cache misses for both approaches are negligible. They resemble exactly the results reported in [1, 29, 32] and references therein for other application areas. Our algorithms' AMR code base Peano [31] relies on a depth-first alike tree traversal [30] that picks up Hölder continuity properties of the Peano space-filling curve

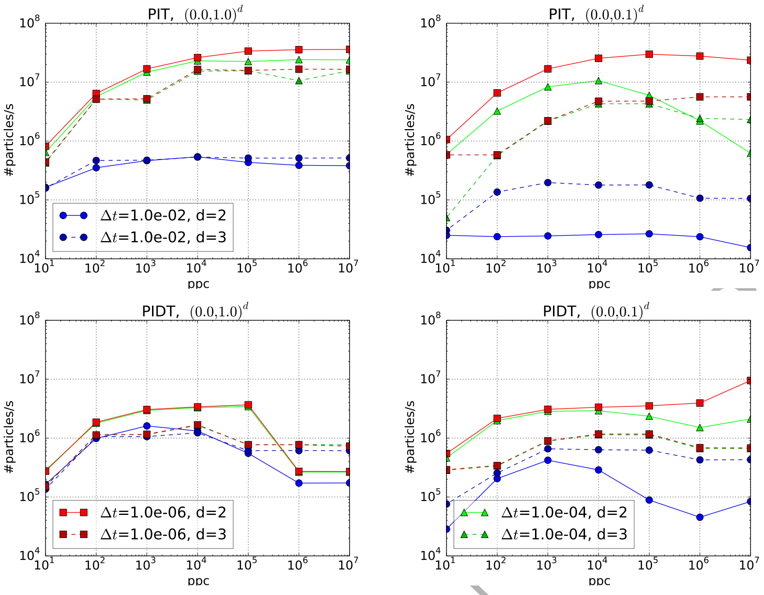


Figure 7: Single core throughput with homogeneous (left) or inhomogeneous/breaking dam (right) start conditions for PIT (top) and PIDT (bottom). $d = 2$ marked by solid lines and $d = 3$ by dashed lines. Figures compare impact of `ppc` choice on throughput for regular (homogeneous) and dynamically adaptive (breaking dam) grids. Results from SuperMUC.

[32]. This implies advantageous spatial and temporal memory access characteristics. However, we expect a reimplementation with another code base to yield advantageous properties as well, as all PIC and PIDT ingredients follow a strict element-wise/local formulation and as the memory subsystem in above measurements is underutilised. A slight increase of cache misses thus does not neccessarily pollute runtime results significantly, if proper prefetching is applied, while the basic memory usage profile is advantageous since all operations are local—either accessing neighbours or parents/children within the grid.

There is a sweet spot from which it pays off to use PIDT rather than PIT. This payoff point depends on time step size and `ppc`. For sufficiently big time steps and reasonable small `ppc`, PIDT outperforms PIT. This is due to the reduction of lift operations, i.e. due to PIT having more particles tunneling. PIT is around a factor of three slower than the pure particle throughput on a single core if Δt is very small. PIDT is around a factor of 3^d slower due to the particle sorting overhead. The remainder of the experiments runs all settings for $\Delta t = 10^{-4}$ as this is an interesting regime where PIT has not yet overtaken PIDT for the majority of experiments. Putting runtime evaluations into relation to the number of lifts always allows us to predict properties of all particle handling algorithms. The remainder of the experiments also restricts to homogeneous settings, i.e. the particles initially are distributed homogeneously among the computational domain, as particle characteristics then remain invariant. Putting runtime evaluations into relation to the number of characteristic ppc allows us to predict properties of simulations where the computational domain comprises different spatial regions with different particle distributions.

6.4. Parallel PIT

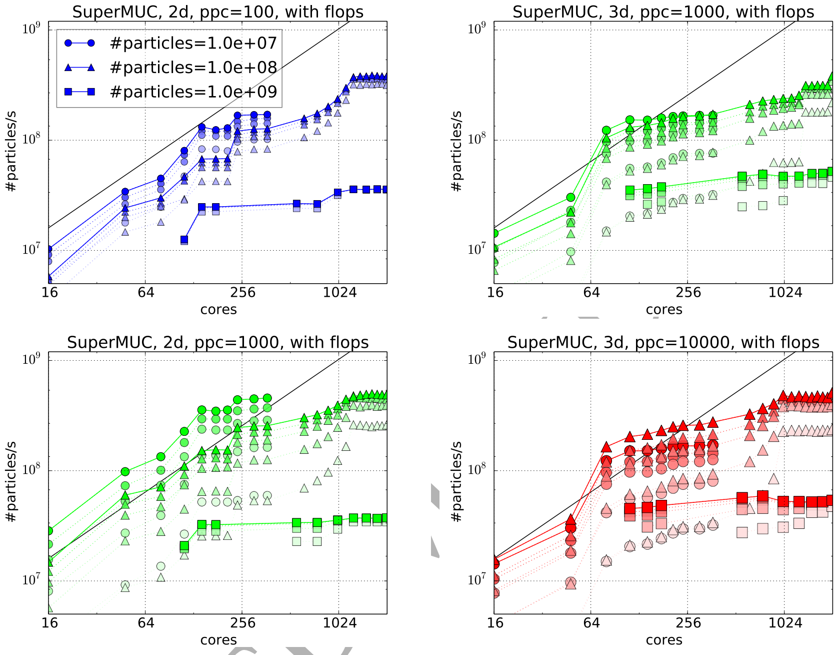


Figure 8: Parallel throughput of PIT. Each plot studies three different total particle numbers (different markers) for one fixed ppc and dimension d. Each particle study is ran five times with 0, 128, 256, 1024 or 4096 flops per particle. The brighter the points the higher the number of flops. The saturated points with a solid line are measurements without any compute flops per particle, i.e. solely the particle move and resort into the grid. Different colours here pick up the colouring from Figure 9 and PIDT experiments.

We start our parallel studies with PIT and artificially perform an additional 0, 128, 256, 1024 or 4096 floating point operations per particle to emulate the solving of a PDE and to be able to study the impact of this additional workload on the particle performance. Each experiment hence was performed five times.

Four properties become apparent from measurements on SuperMUC (Figure 8): First, the more computation is done per particle the lower the throughput, but this difference almost vanishes for high core counts. Second, the throughputs for $d = 2$ and $d = 3$ approach each other for high core counts. Third, PIT's 2d scaling is close to linear up to a given threshold. For $d = 3$, the results are rougher, but a similar threshold is hit for higher core counts. If the core count exceeds this threshold, the throughput stagnates. Fourth, the more particles the lower the parallel efficiency. Eventually, PIT does not scale for 10^9 particles. We observe an inverse weak scaling with respect to particle numbers.

The behaviour results purely from the particle handling, as the synchronisation of the grid along subdomain boundaries is neglectable due to our realisation from [22]. Any master has to wait for all its workers before it may ascend, as the workers might lift particles. Any node triggers its send to its master after the traversal of its local domain has finished. This is a partial synchronisation of the ranks and a blocking data exchange realising a reduction. The latter is sensitive to latency and bandwidth restrictions. As master-worker data exchange prelude or follow the actual local particle handling, the computational work per particle does influence the runtime, as it cannot overlap with the communication. The fewer computation to be done, however, i.e. the more cores participate in the computation, the lower the impact of this work and the more severe communication bounds. As the logical master-worker tree topology is broader for $d = 3$ than for $d = 2$, the synchronisation runtime pressure at the workers is higher for $d = 3$. As latency is critical for the global reduction and its counterpart when we start up the cores and drop particles, latency dominates the runtime if "too" many cores collaborate. The performance then stagnates. As bandwidth is critical for the master-worker data exchange, more particles slow down the reduction phase and its startup counterpart. PIT scales only in a very limited parameter regime.

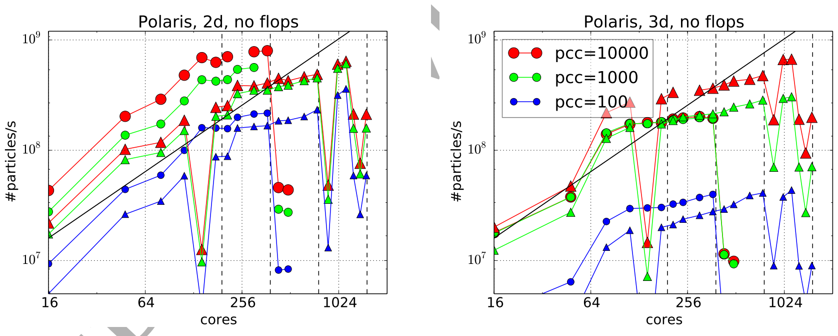


Figure 9: Results from Figure 8 without any additional flops rerun on Polaris. The marker size/colour identifies the ppc, while circles plot 10^7 particles and triangles 10^8 particles. Vertical lines mark whenever the ports of one switch layer theoretically are exceeded. In practice, more switches are used for lower (small) core counts already.

Basically, its scaling is determined by the hardware characteristics plus the tight synchronisation. This effect becomes evident for the same experiments on Polaris (Figure 9). Polaris exhibits slightly better results for moderate core counts. However, the throughput drops whenever the core count requires the supercomputer to employ another level of switches. Due to a good placement that anticipates broken or overbooked nodes, the resulting performance drops appear slightly prior to the theoretical switch capacity. Notably, the more restrictive network topology introduces an inverse weak scaling effect for a high core count to particles ratio, where an increase of cores introduces a degeneration of the throughput due to increased communication/synchronisation pressure.

6.5. Parallel PIDT

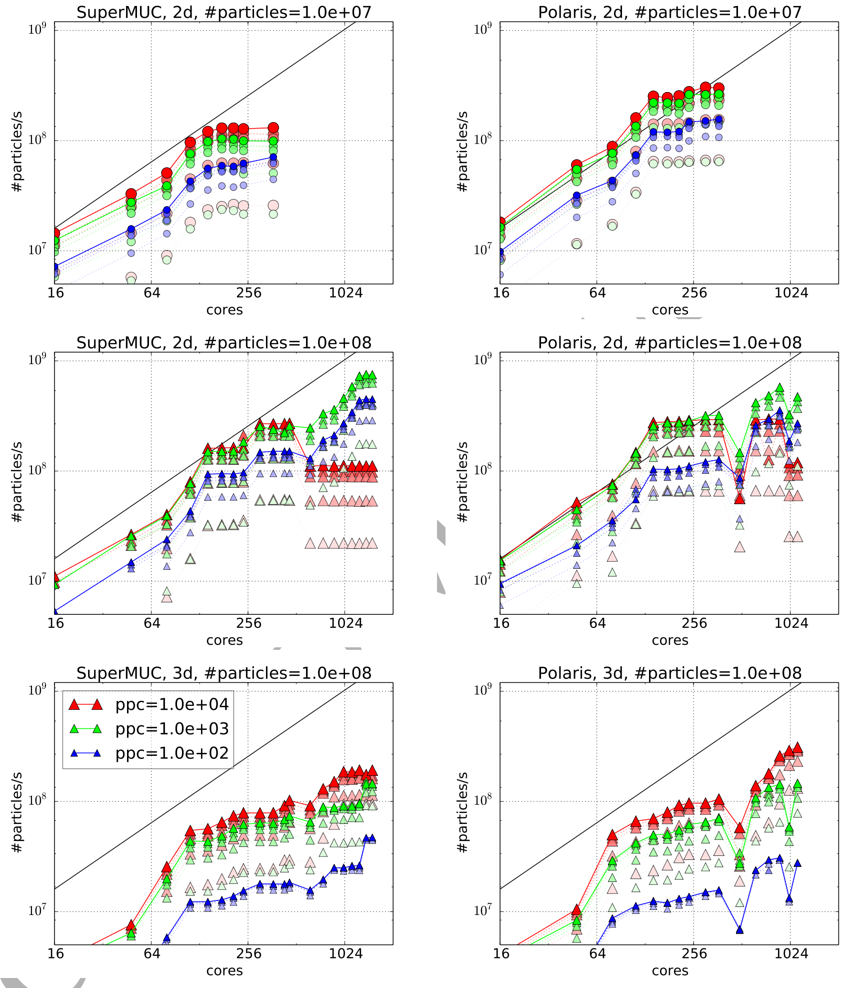


Figure 10: Parallel throughput of PIDT for SuperMUC (left) and Polaris (right). The darker the dots the fewer flops are added to each particle. Solid lines are worst case setups with no computation per particle besides position updates. The bigger the marker the bigger `ppc` $\in \{10^2, 10^3, 10^4\}$ (blue,green,red). All marker types pick up conventions of the other plots.

We next rerun all experiments for PIDT and compare the outcomes to the PIT results. Three differences become apparent (Figure 10): First, the impact of the operations per particle diminishes. Yet, lower arithmetic intensity still means higher throughput. Second, PIDT inherits its lower throughput compared to PIT for low core counts. It still is slower. Third however, PIDT scales better than PIT if the number of particles is reasonably big and ppc is small. There still is a stagnation threshold, but this threshold is higher than for PIT, i.e. PIDT overtakes its sibling algorithm for a decent core count.

The improvement of PIDT compared to PIT stems from the fact that PIDT exchanges particles both via master-worker relations and along subdomain boundaries. The exchange along these boundaries can be realised asynchronously. This allows PIDT to hide computations behind non-blocking MPI calls. Such a hiding is the more effective the more computational workload per particle. However, it also depends on the dimensionality as the domain boundary is a $d-1$-dimensional submanifold. Still, PIDT restricts data along the spacetree each iteration and thus is vulnerable to latency effects. However, the actual number of lifts along the master-worker hierarchy is significantly smaller than for PIT and thus attenuates the impact of bandwidth constraints. Big ppc make the underlying spacetree more shallow and thus counteract to this effect while small ppc lead to deep spacetrees where boundary data exchange and parallel domain handling gain weight in the total runtime profile.

6.6. raPIDT

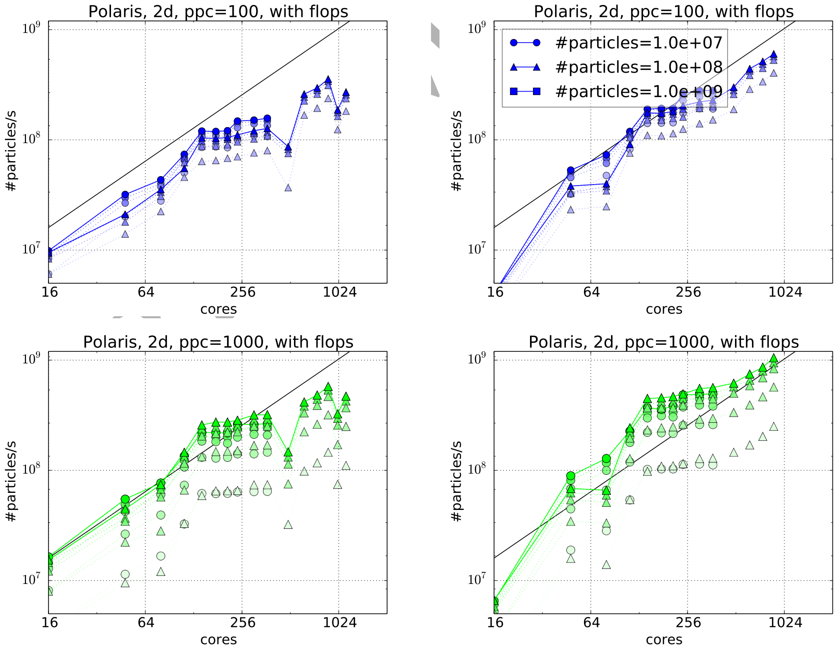


Figure 11: Parallel throughput of PIDT (left) and raPIDT (right) on Polaris.

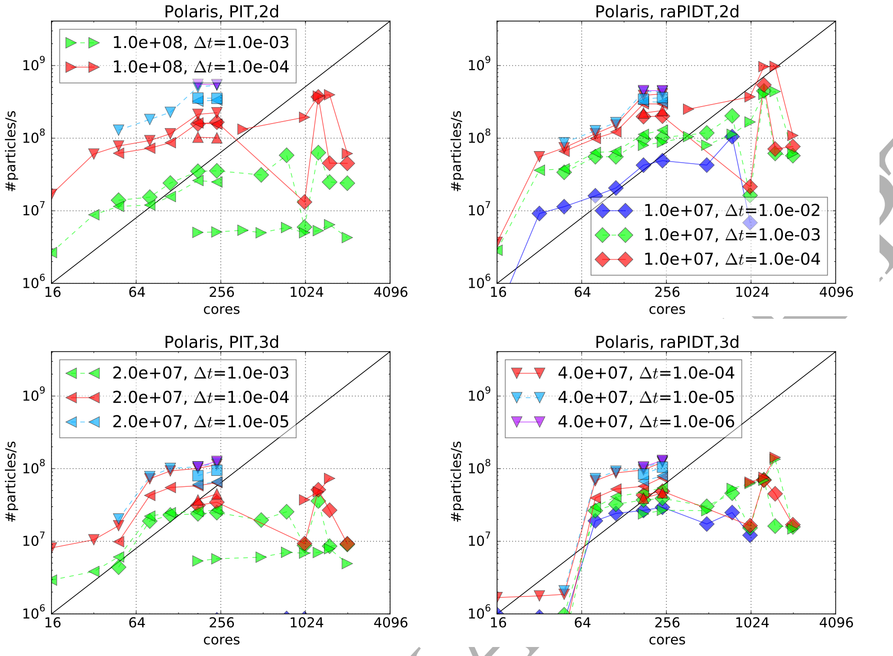


Figure 12: Scaling results for ppc=100 regarding different time step sizes for $d = 2$ (top) and $d = 3$ (bottom). PIT (left) is outperformed by raPIDT (right) while both approaches profit from decreasing time step sizes. Large core count measurements become dominated by network effects that eventually stop any scaling for both approaches on Polaris as opposed to Figure 13.

raPIDT tackles the reduction challenge of our particle management, i.e. skips them if possible, and thus yields improved throughput rates for the previously studied setups (Figure 11) where the time step size is kept constant. It makes the PIDT idea to yield throughput rates comparable to PIT. Notably, it weakens the impact of the network topology.

An interpretation of raPIDT on a bigger core scale requires us to emphasise how the scenario properties normalise experiment parameters. Due to ppc, we control the particle density per spacetree cell, while the total particle count determines the spacetree depth, i.e. the cell sizes. Since we standardise the domain to the unit square or cube, prescribe the time step size and have a uniform velocity distribution, the relative velocity of particles to cell sizes scales with the total particle count by roughly $\sqrt[d]{\#particles}$. An increase of the particle count induces an increase of the particles' velocity. While a study of weak scaling with respect to particles in Section 6.4 and 6.5 is reasonable to understand algorithmic properties, our subsequent performance studies focus on weak scaling where the overall computational domain is successively increased. To enable use to compare results with previous results, we stick with the unit length domain but decrease the time step size when we increase the particle count.

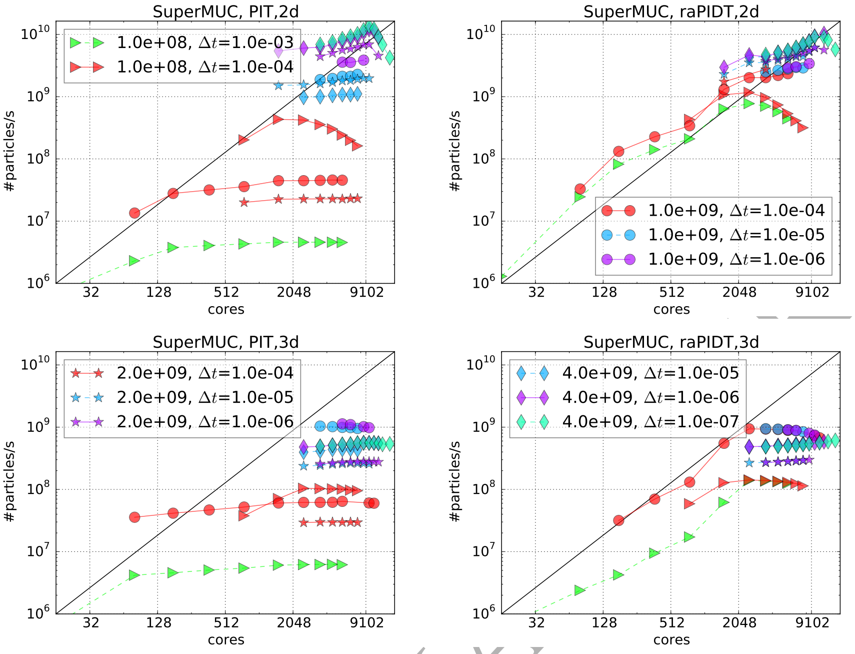


Figure 13: Experiments from Figure 12 for different particle counts rerun on SuperMUC.

Once we apply this normalisation, we observe that raPIDT outperforms PIT (Figures 12 and 13) for reasonable big time step sizes. For small time step sizes (relative to the particle count, i.e. the grid structure), both perform with the same throughput. raPIDT is more robust than PIT on Polaris (Figures 12). The $d = 3$ throughput is lower than the two-dimensional counterpart for both setups, as the typical grid structure for fixed ppc differs. If we normalised with respect to the grid depth, $d = 2$ and $d = 3$ yield similar results. On SuperMUC, the 4:1 blocking topology for more than 8192 cores stops both variants to pass the 10^{10} particles per second threshold and makes the throughput stagnate or degenerate. raPIDT here does not perform significantly more robust than PIT.

7. Comparison with other algorithms

We finally compare PIT, PIDT and raPIDT to alternative implementations. Obviously, only alternatives that support both tunneling and dynamically adaptive grids candidate. The most prominent class of spacetree AMR codes is the family of spacetree algorithms that rely on space-filling curves for the partitioning [1, 8, 15, 20, 21]. Textbook variants of SFC codes typically hold information about their subpartitioning on each rank—they basically store where the SFC's preimage is cut into pieces. Each rank's tree construction starts from its locally owned cells, i.e. the cells in-between the rank's start index and its end index along the SFC. From hereon, the rank constructs the local spacetree in a bottom-up manner: if a cell is held by a rank, also its parent is held by this rank which yields, on coarser levels, a partial replication. Detailed comparisons to the present Peano approach used in our PIT and raPIDT implementation can be found in [30].

If the cuts along the SFC are known, it is possible due to the SFC code [30] to compute per particle the preimage of their position, i.e. an index along the SFC. This immediately yields the information which rank holds a particle. One thus can send any particle directly to the right rank after the move. Though our implementation base relies on a space-filling curve as well [22, 29, 31, 32], it does neither exhibit curve information to the application nor does it hold information about the global partitioning on any rank. However, we can manually expose this information to the ranks and thus reconstruct a SFC-based implementation. This allows us to compare an SFC-based code directly to the present alternatives: Our implementation using SFC cuts runs through the grid and moves the particles as for PIT. Whenever a particle leaves the local domain, we move it into a buffer. For each rank, there's one buffer. Upon termination of the local traversal, all buffers are sent away. As there is no information available whether particles tunnel, each rank then has to wait for a notification from all other ranks. Though this is a global operation, our implementation follows [25] and realises it with point-to-point exchange. If one rank finishes prior to other ranks, the in-between time is then already used for data exchange. Throughout our experiments, any overhead to maintain the global decomposition data is neglected, i.e. we restrict purely to the particle sorting. However, we emphasise that a rank receiving particles does not hold any additional information about the particle position, i.e. where within the local domain it has to be inserted. We thus rely on the drop mechanism. Comparisons to a direct insert based upon SFCs again (the particle's cell is identified due to the preimage and inserted right into the correct cell) show that both variants yield comparable results. This is an important difference to PIT and PIDT as the latter schemes encode remote sorting information implicitly within their send order.

A second competing idea is the sieve approach from [4]. Different to SFCs, it comes along without global domain decomposition knowledge. Each rank collects all particles that leave its domain. Again, we rely on local buffers. These sets of particles then are passed around between the ranks cyclically. Each rank extracts the particles falling into its domain. It sieves and then passes on the array. We distinguish two variants to sort particles into the local grid if we receive a set of particles from the neighbour node: in one variant (sieve & drop), we rely on PIT's drop mechanism. In a second variant, we use the local SFC information to sort the received particles directly into the correct cells.

Besides the alternative variants, we also compare PIT, PIDT and raPIDT to a plain linked-cell algorithm. For the latter, we use a modified PIDT code. The velocity profile is chosen such that no particle may tunnel. As a consequence, we manually switch off all multiscale checks, and we, in particular, skip any master-worker or worker-master communication. Such a test is thus a valid

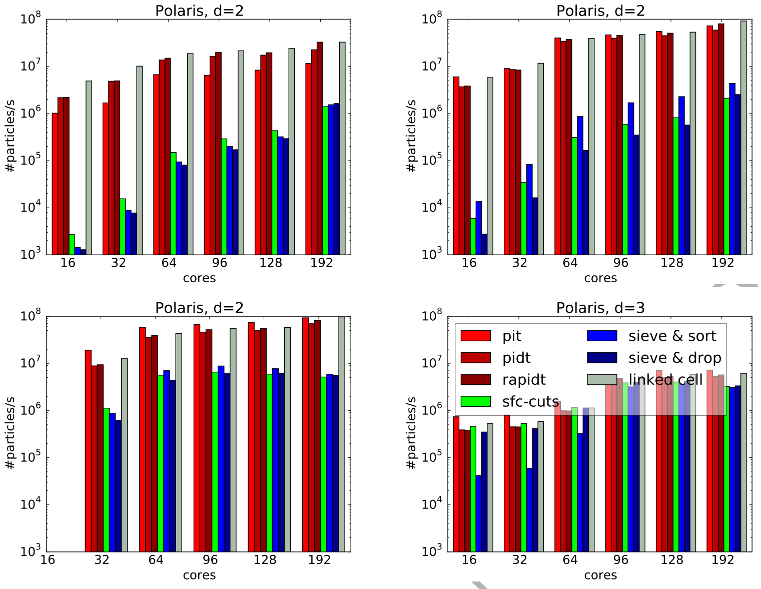


Figure 14: Comparison of different solvers for small core counts and characteristic setups with ppc=100, $\Delta t = 10^{-2}$ (top left), ppc=100, $\Delta t = 10^{-3}$ (top right), ppc=100, $\Delta t = 10^{-4}$ (bottom left). ppc=10, $\Delta t = 10^{-4}$ for $d = 3$ (bottom right) illustrates a regime where all algorithms yield comparable results because of a low particle density. All experiments are ran with 10^6 particles.

setup to quantify the overhead introduced by the tunneling for the present software independent of the quality of implementation. No software ingredient is particular tuned and all codes rely on the same ingredients where possible. All setups rely on exactly the same domain decomposition, i.e. balancing differences are eliminated.

We reiterate that PIDT is faster than PIT for big time step sizes of $\Delta t = 10^{-2}$ (Figure 14). However, each run with the linked-cell approach remains faster though the performance gap closes with increasing core counts. Neither our SFC-cut nor our sieve implementations can cope with these results. However, they always scale better. For sufficiently big Δt, they eventually close the performance gap. If we reduce the time step size, there is a sweet spot where PIT becomes faster than PIDT as well as the linked-cell approach (visible here only for $d = 3$). Still, the sieve algorithm yields significantly lower throughput. The SFC-based variants are slower, too.

The differences between PIT and PIDT have been discussed already. Our experiments validate, for most setups, statements from [4] that observe that all variants supporting tunneling are slower than a pure linked-cell like code. However, both PIT and PIDT reduce this runtime difference significantly. This is insofar interesting, as PIT exhibits similarities to the `up_down_tree` algorithm from [4] which is reported to be even slower than sieve. Sieve suffers from a global communication phase after each iteration where the length of the phase (number of data exchange steps) increases with more ranks participating. It thus cannot scale. PIT circumnavigates such a global data exchange phase. While the SFC-cut implementation scales and allows all the ranks to process their local grid asynchronously, it still runs into a synchronisation phase, as each rank has to wait per time step for every other rank for notification. This point-to-point synchronisation determines the performance, since we found no significant runtime difference when we studied whether a drop mechanism for local sorting or the direct application of SFC indices yields better results for SFCs. PIT and PIDT are able to spread all data transfer more evenly accross the executation time than their competitors. It is in particular interesting that PIT manages to outperform the linked-cell approach for very small time steps and $d = 3$ with small ppc, i.e. a deep spacetree. This results from the fact that particles within a cell are not sorted and few particles cross the cell faces per time step. Obviously this is an unfair competition: as no neighbourhood information is available in PIT (different to PIDT where we have half a cell overlap), no particle-particle interaction can be realised. PIT is designed for particle-mesh interaction only. Our SFC implementation was able to compete with PIT and PIDT only for small core counts and deep spacetree hierarchies. For bigger core counts, the SFC's behaviour resembles the sieve algorithm which results from the global communication phase found there, too.

We emphasise that the whole runtime picture might change if the grid were regular, if the grid were not fixed prior to the particle sorting, i.e. if we could create the grid depending on the particles without persistent grid data, or if we translated our reduction-avoiding mechanism from raPIDT into the SFC world. The latter would yield a higher asynchronity level and remove a global SFC communication phase.

8. Conclusion and outlook

We introduce three particle management variants facilitating solvers that have to resort particles into an existing dynamically adaptive Cartesian grid frequently and can run into tunneling. Particle in tree (PIT) holds the particles within the tree, i.e. the cells of a multiscale adaptive Cartesian grid, and lifts and drops the particles between the levels to move particles in-between cells. Particle in dual tree (PIDT) stores the particles within the vertices, i.e. switches to a dual grid, and thus fuses ideas of a tree-based particle management with a multiscale linked-list paradigm. Particles either can move up or down within the (dual) tree or directly into neighbouring cells. The latter works on any level of the multiscale grid. raPIDT finally augments PIDT by a simple velocity analysis. This analysis labels regions where no particle is moved in-between certain grid levels in the subsequent time steps. This allows us to eliminate reductions within the tree locally.

Comparisons of the straightforward PIT with the dual grid strategy PIDT reveal that no scheme is superior to the other schemes by default. For small problem setups with slowly moving particles, PIT yields higher performance than PIDT. For big problem setups or fast particles, there is a sweet spot where the higher code complexity of PIDT pays off. Particular interesting is the fact that raPIDT picks up advantages of linked cell strategies, i.e. asynchronity of the ranks and exchange of data in the background, while arbitrary tunneling still is enabled. This might make it a promising candidate for the upcoming massively parallel age. It is also not surprising that the particle throughput depends on differences in the hardware characteristics. High clock rates pay off for moderate particle counts. For high core counts, topology properties of the interconnection network outshine clocking considerations. Regarding latency effects, raPIDT is more robust than PIT and PIDT. For big core counts and high particle numbers, all variants however continue to suffer from latency and bandwidth constraints introduced by x:1 blocking.

Our particle maintenance strategies are of relevance for multiple particle-in-cell (PIC) simulations as well as other particle-grid codes with different algorithmic properties. Examples for PIC are plasma processes with shocks or reconnection, global large-scale simulations with multifaceted particle velocity characteristics, or self-gravitating systems whose dynamics imply large particle inhomogeneities. While such setups require substantial investment into the PDE solver, our particle treatment transfers directly. Another example is local particle time stepping where some particles march in time fast and thus might tunnel, while others then follow up with tiny time steps where the reduction skips come into play. It is part of our future work to use the present algorithms in such a context. Of particular interest to us is the fusion with applications that realise implicit schemes or particle-particle interactions. While the former weaken the grid size and, hence, time step constraints—though we still can make the grid size adapt to fine-scale inhomogeneities of the PDE without algorithmic constraints resulting from the particle velocities—both increase the arithmetic intensity and thus damp the impact of communication on the scaling. We further recognise that the dual grid approach allows us to realise any interaction with a cut-off radius smaller than half the grid width without any additional helper data structure such as linked-cell lists or modified linked lists [7, 17]. In this sense, our approach is related to dual tree traversals [3, 28] that also avoid to build up connectivity links.

It seems to be important to stress two facts. On the one hand, our lift and drop mechanisms can also be used by particle-interaction kernels to push particles to the “right” resolution level, i.e. a level fitting to their cut-off radius. For complex applications, there is no need to drop particles always into the finest grid all the time. This enables particles suspended within the tree. It is subject of future work with respect to applications with particle-particle interaction to exploit such a multilevel storage. On the other hand, we appreciate how fast linked-list algorithms perform in many applications. In our formalism, they basically induce an overlapping domain topology on top of a given tree distribution. And we recognise that the fastest variants of these algorithms rely on incremental updates of these interaction lists. It is hence a straightforward idea to combine our grammar paradigm plus the particle management with an update of these lists where the grammar eliminates also list updates.

Acknowledgements

The authors gratefully acknowledge the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (LRZ, www.lrz.de), as well for the support of the LRZ. This work also made use of the facilities of N8 HPC provided and funded by the N8 consortium and EPSRC (Grant No. N8HPC_DUR_TW_PEANO). The Centre is co-ordinated by the Universities of Leeds and Manchester. Special thanks are due to Bram Reps and Kristof Unterweger for their support on linear algebra and implementation details. Kristof also contributed to make the numerical experiments run. Robert Glas and Stefan Wallner made a major contribution to the maturity and performance of the presented implementations as they used it for their astrophysics code and thus pushed the development. Bart Verleye was funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). All underlying software is open source and available at [31].

References

[1] M. Bader. *Space-Filling Curves - An Introduction with Applications in Scientific Computing*, volume 9 of *Texts in Computational Science and Engineering*. Springer-Verlag, 2013.

[2] C. K. Birdsall and A. B. Langdon. *Plasma physics via computer simulation.* Taylor and Francis, first edition, 2005.

[3] W. Dehnen. A hierarchical o(n) force calculation algorithm. *J. Computational Physics*, 179(1):27–42, 2002.

[4] A. Dubey, K. Antypas, and C. Daley. Parallel algorithms for moving Lagrange data on block structured Eulerian meshes. *Parallel Computing*, 37:101–113, 2011.

[5] M. Durand, B. Raffin, and F. Faure. A Packed Memory Array to Keep Moving Particles Sorted. In J. Bender, A. Kuijper, D. W. Fellner, and E. Guerin, editors, *Workshop on Virtual Reality Interaction and Physical Simulation*. The Eurographics Association, 2012.

[6] F. Fleissner and P. Eberhard. Parallel load-balanced simulation for short-range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection. *Int. J. Numer. Methods Eng.*, 74(4):531–553, 2008.

[7] P. Gonnet. A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations. *J. Comput. Chem.*, 28(2):570–573, 2007.

[8] T. Hamada, T. Narumi, R. Yokota, K. Yasuoka, K. Nitadori, and M. Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 62:1–62:12, New York, NY, USA, 2009. ACM.

[9] R. Hockney and J. Eastwood. *Computer Simulation Using Particles*. Academic Press, 1988.

[10] M. Ihmsen, N. Akinci, M. Becker, and M. Teschner. A parallel sph implementation on multi-core cpus. *Comput. Graph. Forum*, 30(1):99–112, 2011.

[11] D. E. Knuth. The genesis of attribute grammars. In P. Deransart and M. Jourdan, editors, *WAGA: Proceedings of the international conference on Attribute grammars and their applications*, pages 1–12. Springer-Verlag, 1990.

[12] N.A. Krall and A.W. Trivelpiece. *Principles of Plasma Physics*. McGraw-Hill, 1973.

[13] G. Lapenta. Particle simulations of space weather. *Journal of Computational Physics*, 231(3):795–821, 2012.

[14] G. Lapenta and J.U. Brackbill. Control of the number of particles in fluid and mhd particle in cell methods. *Computer Physics Communications*, 87(1–2):139–154, 1995.

[15] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. *Commun. ACM*, 55(5):101–109, 2012.

[16] S. Li and W. K. Liu. Meshfree and particle methods and their applications. *Appl. Mech. Rev.*, 55:1–34, 2002.

[17] W. Mattson and B. M. Rice. Near-neighbor calculations using a modified cell-linked list method. *Computer Physics Communications*, 119(2-3):135–148, 1999.

[18] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995.

[19] S. J. Plimpton, D. B. Seidel, M. F. Pasik, R. S. Coats, and G. R. Montry. A load-balancing algorithm for a parallel electromagnetic particle-in-cell code. *Computer Physics Communications*, 152(3):227 – 241, 2003.

[20] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[21] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros. Dendro: parallel algorithms for multigrid and amr methods on 2:1 balanced octrees. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 18:1–18:12, Piscataway, NJ, USA, 2008. IEEE Press.

[22] M. Schreiber, T. Weinzierl, and H.-J. Bungartz. Sfc-based communication metadata encoding for adaptive mesh. In M. Bader, editor, *Proceedings of the International Conference on Parallel Computing (ParCo)*, volume 25 of *Advances in Parallel Computing: Accelerating Computational Science and Engineering (CSE)*. IOS Press, October 2013. accepted.

[23] H. Shamoto, K. Shirahata, A. Drozd, H. Sato, and S. Matsuoka. Large-scale distributed sorting for gpu-based heterogeneous supercomputers. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 510–518, 2014.

[24] V. Springel. E pur si muove: Galilean-invariant cosmological hydrodynamical simulations on a moving mesh. *NMRAS*, 401:791–851, 2010.

[25] H. Sundar, D. Malhotra, and G. Biros. Hyksort: A new variant of hypercube quicksort on distributed memory architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 293–302, New York, NY, USA, 2013. ACM.

[26] J. Treibig, G. Hager, and G. Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 207–216. IEEE Computer Society, 2010.

[27] B. Verleye, P. Henri, R. Wuyts, G. Lapenta, and K. Meerbergen. Implementation of a 2D electrostatic particle in cell algorithm in Unified Parallel C with dynamic load-balancing. *Computers & Fluids*, 80:10–16, 2013.

[28] M. S. Warren and J. K. Salmon. A portable parallel particle program. *Computer Physics Communications*, 87:266–290, 1995.

[29] T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids.* Verlag Dr. Hut, München, 2009.

[30] T. Weinzierl. The Peano software—parallel, automaton-based, dynamically adaptive grid traversals. Technical Report 2015arXiv150604496W, Durham University, 2015.

[31] T. Weinzierl et al. Peano—a Framework for PDE Solvers on Spacetree Grids, 2012. www.peano-framework.org.

[32] T. Weinzierl and M. Mehl. Peano – A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids. *SIAM Journal on Scientific Computing*, 33(5):2732–2760, October 2011.

Appendix A. Validation

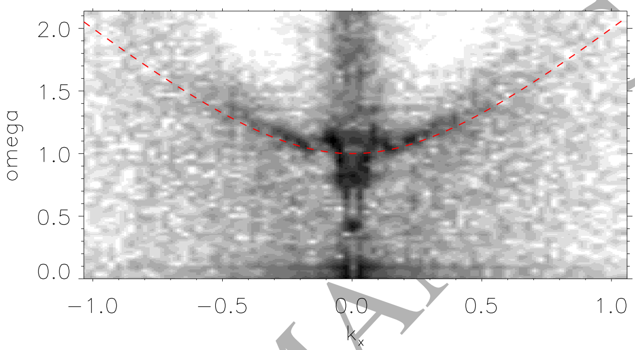


Figure A.15: Electric field spectrum in the wavevector-frequency plane (k_1, ω), at $k_2 = 0$. The colors range from white (low signal) to black (high signal). The red dashed line shows the theoretical Langmuir wave dispersion relation obtained from the Vlasov-Poisson theory.

As validation of our PIC implementation—which is independent of the choice of PIT or PIDT—we study a thermal plasma at rest, and we confirm that our code retrieves the theoretical Langmuir wave dispersion relation obtained from the Vlasov-Poisson theory. For our test, we rely on a regular $d = 2$ grid with 243×243 grid cells. Time is normalized to the inverse plasma frequency ω_p^{-1}, and space is normalized to the Debye length λ_d, i.e. a grid cell has size $\lambda_d \times \lambda_d$. The (angular) plasma frequency is defined by

$$\omega_p = \sqrt{\frac{ne^2}{\epsilon_0 m_e}}$$

with n being the electron density, m_e being the electron mass and e being the elementary charge. The Debye length λ_d, the typical length scale for the electric screening of a charge in a plasma, is defined by

$$\lambda_d = \sqrt{\frac{\epsilon_0 k_B T}{ne^2}} = v_{th}/\omega_p$$

with ϵ_0 being the vacuum permittivity, k_B being the Boltzmann constant, and T being the electron temperature. All parameters follow [12]. Our observed

Langmuir waves, also called electron plasma waves, are high frequency oscillations of the electron density over a fixed ion background. The frequency is high enough for the ions not to have time to respond because of their higher inertia. The charge separation generates an electric force acting as the restoring force.

In the cold plasma approximation, i.e. when the thermal velocity of the plasma is much smaller than the phase velocity of the wave, the Langmuir waves oscillate at the plasma frequency. When taking into account the finite temperature of the electrons, a (small) frequency correction appears, leading to the following dispersion relation for a Maxwellian velocity distribution function

$$\omega^2 = \omega_p^2 + 3k^2 v_{th}^2 = \omega_p^2(1 + 3k^2\lambda_d^2)$$

where k_L is the wave vector, v_{th} the electron thermal velocity.

The electron plasma is initially loaded with 100 macro-particles per cell, a charge-to-mass ratio $q/m = -1$, and a random velocity characterized by a Gaussian distribution of mean velocity 0 and thermal velocity $v_{th} = 1$. The electron charge density is initially uniform $n = -1$. We also set a neutralising fixed ion background—this contribution adds as constant term to the right-hand side in (1) besides the simulated particles—with a constant charge density $n_i = 1$. Our simulation runs with a time step size $\Delta t = 0.01$ and the potential is output every 150 time steps.

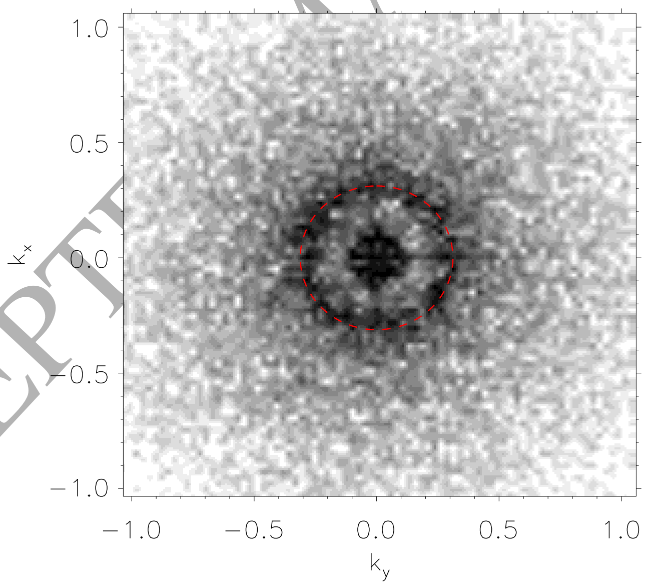


Figure A.16: Electric field spectrum in the wavevector-wavevector plane (k_1, k_2), at $\omega = 1.14$. The colors range from white (low signal) to black (high signal). The red dashed line shows the wavevectors in the $d = 2$ plane corresponding to Langmuir waves at frequency $\omega = 1.14$, from the theoretical Langmuir wave dispersion relation obtained from the Vlasov-Poisson theory.

The initial random velocity seeds a sea of Langmuir waves whose dispersion relation can be checked from the analysis of the electric potential: the output electric potential $V(x,y,t)$ is Fourier transformed both in time and space which yields $\hat{V}(k_1,k_2,\omega)$ where k_1 and k_2 are the wave vectors associated to the space coordinates x_i $i \in \{1,2\}$. ω is the frequency. The Fourier transform of the electric field is shown for a cut at $k_y = 0$ in the plane (k_x,ω) in Figure A.15. The electric fluctuations are organized in Fourier space so to match the theoretical Langmuir wave dispersion relation obtained from the Vlasov-Poisson theory $\omega^2 = \omega_p^2 \left(1 + 3\,(k\lambda_d)^2\right)$ (dashed line). The result is identical in a cut at $k_x = 0$ in the plane (k_y,ω). The Fourier transform of the electric field is also shown in a two-dimensional cut at fixed frequency $\omega = 1.14$, in the wavevectors plane (k_1,k_2), in Figure A.16. Again, the electric fluctuations are organized in Fourier space at the wave vectors corresponding to Langmuir waves (dashed line) at the prescribed frequency.

Appendix B. Additional experimental data

Table B.2: Experiments from Table 1 reran on Polaris.

p	1	2	4	8	12	16	32
10^3	$4.59 \cdot 10^7$	$4.86 \cdot 10^7$	$\mathbf{5.26 \cdot 10^7}$	$3.73 \cdot 10^7$	$2.94 \cdot 10^7$	$3.03 \cdot 10^7$	$1.92 \cdot 10^7$
10^4	$1.04 \cdot 10^8$	$1.65 \cdot 10^8$	$2.24 \cdot 10^8$	$\mathbf{2.36 \cdot 10^8}$	$2.24 \cdot 10^8$	$1.98 \cdot 10^8$	$1.32 \cdot 10^8$
10^5	$1.17 \cdot 10^8$	$2.31 \cdot 10^8$	$4.18 \cdot 10^8$	$6.60 \cdot 10^8$	$8.08 \cdot 10^8$	$\mathbf{8.83 \cdot 10^8}$	$7.04 \cdot 10^8$
10^6	$1.14 \cdot 10^8$	$2.35 \cdot 10^8$	$4.57 \cdot 10^8$	$8.52 \cdot 10^8$	$1.20 \cdot 10^9$	$\mathbf{1.38 \cdot 10^9}$	$1.33 \cdot 10^9$
10^7	$1.11 \cdot 10^8$	$2.20 \cdot 10^8$	$4.26 \cdot 10^8$	$5.99 \cdot 10^8$	$6.24 \cdot 10^8$	$\mathbf{6.29 \cdot 10^8}$	$5.93 \cdot 10^8$
10^8	$1.05 \cdot 10^8$	$2.20 \cdot 10^8$	$4.30 \cdot 10^8$	$6.02 \cdot 10^8$	$6.28 \cdot 10^8$	$\mathbf{6.31 \cdot 10^8}$	$6.26 \cdot 10^8$
10^3	$\mathbf{5.92 \cdot 10^7}$	$4.52 \cdot 10^7$	$3.91 \cdot 10^7$	$3.26 \cdot 10^7$	$3.01 \cdot 10^7$	$3.40 \cdot 10^7$	$1.90 \cdot 10^7$
10^4	$7.85 \cdot 10^7$	$1.28 \cdot 10^8$	$1.83 \cdot 10^8$	$\mathbf{2.13 \cdot 10^8}$	$1.94 \cdot 10^8$	$2.00 \cdot 10^8$	$1.18 \cdot 10^8$
10^5	$9.77 \cdot 10^7$	$1.90 \cdot 10^8$	$3.40 \cdot 10^8$	$5.64 \cdot 10^8$	$6.82 \cdot 10^8$	$\mathbf{7.20 \cdot 10^8}$	$5.42 \cdot 10^8$
10^6	$8.94 \cdot 10^7$	$1.71 \cdot 10^8$	$3.36 \cdot 10^8$	$5.53 \cdot 10^8$	$6.44 \cdot 10^8$	$\mathbf{6.71 \cdot 10^8}$	$6.13 \cdot 10^8$
10^7	$7.75 \cdot 10^7$	$1.67 \cdot 10^8$	$3.20 \cdot 10^8$	$4.05 \cdot 10^8$	$4.17 \cdot 10^8$	$\mathbf{4.20 \cdot 10^8}$	$4.07 \cdot 10^8$
10^8	$7.73 \cdot 10^7$	$1.72 \cdot 10^8$	$3.22 \cdot 10^8$	$4.07 \cdot 10^8$	$4.19 \cdot 10^8$	$\mathbf{4.24 \cdot 10^8}$	$4.21 \cdot 10^8$

We ran the stream-like benchmark without particle sorting or any grid on both Polaris and SuperMUC. On Polaris, it yields almost twice the performance compared to SuperMUC (Table B.2). This is a direct result of the higher clock frequency on the smaller cluster.

Finally, some additional plots illustrating the lift behaviour for a fixed time step size can be found in Figure B.17. They validate our statements on pros and cons of PIT or PIDT respectively.

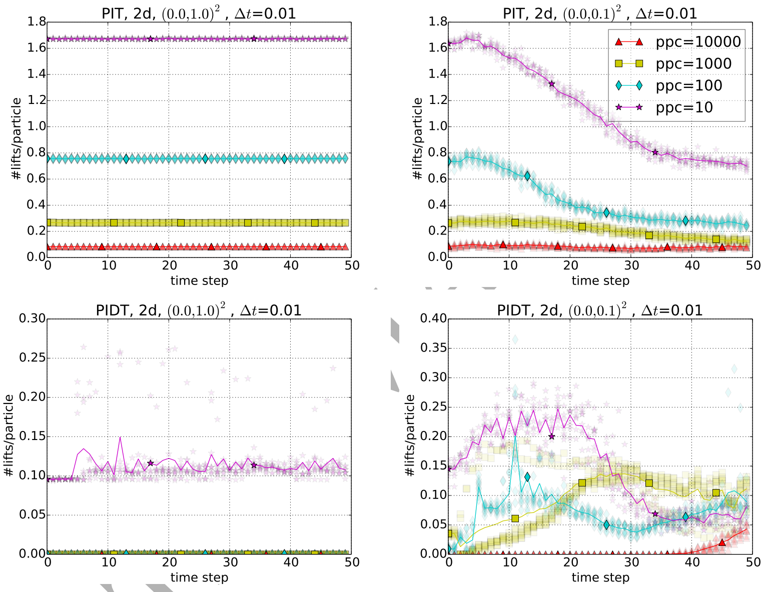


Figure B.17: $\Delta t = 0.01$ for PIT (top) and PIDT (bottom). We restrict to $d = 2$ and study homogeneous (left) and inhomogeneous (right) initial distributions, i.e. regular and dynamically adaptive grids.