Comput Mech (2010) 46:103–114
DOI 10.1007/s00466-009-0436-x

ORIGINAL PAPER

The PDE framework Peano applied to fluid dynamics: an efficient implementation of a parallel multiscale fluid dynamics solver on octree-like adaptive Cartesian grids

Hans-Joachim Bungartz · Miriam Mehl · Tobias Neckel · Tobias Weinzierl

Received: 17 September 2009 / Accepted: 13 October 2009 / Published online: 4 November 2009

Abstract This paper presents the general purpose framework Peano for the solution of partial differential equations (PDE) on adaptive Cartesian grids. The strict structuredness and inherent multilevel property of these grids allows for very low memory requirements, efficient (in terms of hardware performance) implementations of parallel multigrid solvers on dynamically adaptive grids, and arbitrary spatial dimensions. This combination of advantages distinguishes Peano from other PDE frameworks. We describe shortly the underlying octree-like grid type and its most important properties. The main part of the paper shows the framework concept of Peano and the implementation of a Navier–Stokes solver as one of the main currently implemented application examples. Various results ranging from hardware and numerical performance to concrete application scenarios close the contribution.

Keywords PDE framework · Octree-like grids · Cartesian grids · Computational fluid dynamics · Moving geometries · Multigrid methods · Parallelisation · Memory efficiency

H.-J. Bungartz · M. Mehl (✉) · T. Neckel · T. Weinzierl
Technische Universität München, Boltzmannstr. 3,
85748 Garching, Germany
e-mail: mehl@in.tum.de

H.-J. Bungartz
e-mail: bungartz@in.tum.de

T. Neckel
e-mail: neckel@in.tum.de

T. Weinzierl
e-mail: weinzier@in.tum.de

1 Introduction

Methods to treat partial differential equations (PDE) numerically are part of the foundations of many disciplines in science and engineering. The progress in these disciplines, with computational fluid dynamics (CFD) among them, relies on a permanently improving software base—software that facilitates the realisation of sophisticated algorithms, software that is able to handle bigger and bigger data sets as well as systems of equations, and software that makes the most of the hardware of supercomputers. Within a CFD project, developing such high-end codes from scratch is usually impossible due to a lack of development time and due to a lack of experience in the individual fields such as numerical methods, hardware-aware algorithms, and software engineering. As a result, more and more CFD codes are based on frameworks, i.e. code ecosystems that provide several tools and features (e.g. visualisation, data import, parallelisation), provide a clear encapsulation and separation-of-concerns design, and provide an environment for the interplay, exchangeability, and integration of individual application parts.

This paper describes our framework Peano. As a use case, we apply it to CFD. Peano is a general-purpose environment for matrix-free PDE solvers on adaptive Cartesian grids. It brings together hardware-efficiency paradigms with state-of-the-art numerical methods, it is able to handle large numbers of degrees of freedom, and it provides a rigourous separation-of-concerns design. In this paper, we introduce our framework from a CFD point of view, i.e. we focus on the Navier–Stokes equations implemented within the framework and the framework design and paradigm itself.

Peano is not the first general-purpose code (see [4,19]). However, its focus on exploiting the full potential of supercomputers in combination with sophisticated numerical

methods is, to our knowledge, rather unique. This combination is possible, as the whole framework relies on adaptive Cartesian grids induced by k-spacetrees—a generalisation of the well-known quad- and octrees. They are traversed along a space-filling curve. What sounds like a restriction (other frameworks provide a zoo of different grids) proves of value. Concentrating on Cartesian grids simplifies, first, the coding of the solver. Second, the focus on this structured grid type and the fact that the framework's solvers are matrix-free allow Peano to come along with a very small memory footprint: One bit per unknown is sufficient to store a dynamically adaptive grid, and the solver exhibits a high cache hit rate. Third, features such as the support for arbitrary dimensions, domain decomposition, dynamical load balancing, geometric multigrid solvers, and multicore support fit seamlessly into the concept—the major insight of [32]. Fourth, instead of an iterator to traverse the grid, Peano implements an event-driven concept: Its PDE solvers plug into the traversal due to call-back points (events) and implement there their PDE-specific actions. Thus, traversal, parallelisation, grid storage, and PDE-specifics, are separated from each other, while the individual components properties intertwine for the resulting solver—a property first studied in [23] for a computational fluid dynamics solver.

The remainder is organised as follows: In Sect. 2, we present the underlying spacetree grids, their traversal, Peano's data management, and the framework's unique selling points. The framework concept itself is described in Sect. 3. Section 4 introduces the implementation of a fluid dynamics solver. Several results are given in Sect. 5. While the paper presents our particular software environment, which is the one primary aim of this text, these results show that fluid dynamics profit from adaptive Cartesian grids. The framework opens, due to its unique selling points, the opportunity to tackle new challenges not manageable before. A short conclusion with an outlook close the discussion.

2 Spacetrees and adaptive Cartesian grids

Peano is a framework for solvers on adaptive Cartesian grids. These grids stem from k-spacetrees which are a generalisation of the well-known quad- and octree idea looking back to a long tradition in computational sciences and engineering [8,22,25] and are more and more frequently used for efficient implementations of computational methods in data management, computer graphics, engineering, and numerical simulation ([2,3,5,17,18,32] and references therein). To construct a grid for an arbitrary computational domain, we embed the domain into a hypercube. This hypercube is cut equidistantly into a fixed number of k pieces along each coordinate axis. The result is k^d smaller hypercubes. We continue recursively until a (local) termination criterion is met. The actual geometry of the computational domain is defined in a marker-and-cell like manner [14,28]. The overall structure of the grid equals a tree (Fig. 1).

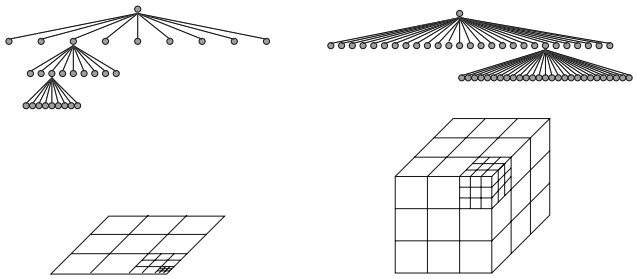

Fig. 1 Examples for two-dimensional and three-dimensional ($k = 3$)-spacetree grids. Above the grids, the corresponding trees of grid cells are shown

The k-spacetree construction process yields a sequence of finer and finer adaptive Cartesian grids where hanging nodes occur at the interface of cells refined further and cells not refined further. It is very easy to refine or coarsen the spacetree locally: just refine individual leaves or remove subtrees, respectively. Furthermore, the spacetree yields a multiscale representation of the grid, as the tree comprises all the coarser geometric cells, too.

For grid-based numerical simulations, spacetrees offer a lot of advantages due to their strict structuredness: fast and straightforward grid generation, support for dynamical refinement as well as coarsening, a grid hierarchy for multilevel and multigrid algorithms, and one can, finally, encode them very economically as we discuss in a moment. Such, they have established as an attractive alternative to unstructured grids, e.g. (see for example [10,26,27,29]). With matrix-free algorithms, these low memory and flexibility arguments hold for any solver implemented on the spacetree, as no additional data structures impose additional overhead in terms of memory and maintenance work. However, the spacetree traversal then has to be fast: It is the lynchpin of the solver since each operation on the grid (such as a multigrid smoothing operation) has to traverse the grid data structure.

Our framework provides solely one traversal type implemented by an automaton. This traversal follows the depth-first idea, i.e. it runs through the spacetree along a depth-first order. Such a classical backtracking-like processing equals an element-wise grid traversal, i.e. Peano supports strict element-wise algorithms exclusively. While the traditional quadtree equals the parameter combination $d = 2, k = 2$, and while an octree equals $d = 3, k = 2$, Peano uses ($k = 3$)-spacetrees. The reason for that choice is the Peano space-filling curve [24]: It is used to make our traversal definition deterministic. So far, there is no order defined on the k^d children of any refined cell. Here, we apply the space-filling curve to sort them (Fig. 2), which allows for efficient data structures and data access patterns. This important additional

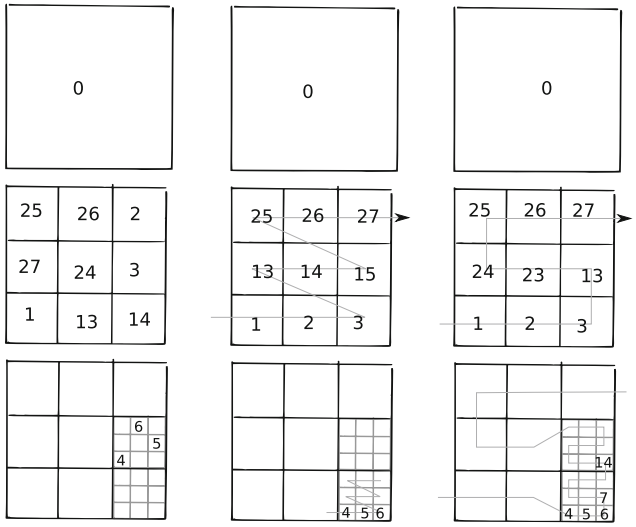

Fig. 2 A depth-first traversal of a spacetree (here $d = 2$ and $k = 3$) is non-deterministic as the children of a refined node have no order, i.e. a traversal has to be made deterministic due to an order on the children. *Left* arbitrary enumeration; *middle* lexicographic enumeration; *right* enumeration along the iterates of the Peano curve

effect could not be achieved with any other curve such as the Hilbert curve.

With a deterministic element-wise traversal at hand, PDE solvers plug into the automaton's transitions to implement the solver behaviour directly on the grid. The grid storage and the traversal implementation are hidden from the PDE programmer, and properties of the traversal realisation—its performance, its low memory requirements, its parallelisation, e.g.—carry directly over to the PDE solver. From our point of view, the following properties [32] are especially outstanding:

First, the k-spacetree can be applied for any dimension $d \geq 2$. In the code, d is just a compile-time parameter. For the parabolic problems discussed here, $d \in \{2, 3, 4\}$ is of value with the fourth dimension representing time in a space–time adaptive approach.

Second, the tree structure of the k-spacetree grids naturally translates to multilevel data stored on the grid which is useful for the implementation of multiscale solvers—in particular geometric multigrid algorithms—without losses in hardware efficiency compared to single level solvers.

Third, each refinement step of the k-spacetree construction recursively splits up grid cells into k^d smaller, disjoint cells. This property is used in Peano to implement a parallel recursive construction of a non-overlapping domain decomposition and a distributed dynamical load balancing.

Fourth, each refined cell of the k-spacetree holds exactly k^d children. Operator evaluations on these children can run in parallel. Peano's multicore extension exploits this property. In addition, it unrolls several recursion levels wherever patches with equal refinement depth are detected.

Fifth, due to the strict structuredness of k-spacetree grids, due to the uniquely defined depth-first traversal order along the Peano curve, and due to the cell-wise operator evaluation, one bit per spacetree node (refined or not refined) is sufficient to store the whole grid information.

Sixth, the whole data management during a grid traversal uses a fixed number of $2 \times d + 2$ stacks. As any sequence of memory accesses to a small, fixed number of stacks—which possess a typical size of several megabytes only—exhibits a high spatial and temporal locality, a high cache hit-rate is the consequence.

Seventh, it is easy to add additional leaves to the k-spacetree or to remove complete subtrees without affecting the efficiency of the data structures and data access algorithms. Consequently, all Peano plugins are able to update the adaptive grid seamlessly throughout the solving process. This, of course, holds in particular for matrix-free solvers that do not have to update any global system matrices.

A seemingly disadvantages of Cartesian grids is that they approximate arbitrary domain boundaries with $\mathcal{O}(h)$ accuracy, only, while almost all operator discretisation approaches exhibit at least second order accuracy. Peano compensates this with a local adaptivity coming along with a very low memory overhead. Of course, this grid adaptivity has to enhanced with a more sophisticated boundary treatment such as cut-cell [12], immersed boundary [21], or extended finite element [9,13] approaches in the future. Structured but still adaptive Cartesian grids such as spacetree grids come along with very low memory demands and no assembly overhead if combined with matrix-free solvers. However, only with this restriction to matrix-free solvers and the tight connection between the grid, traversal algorithms, and solver numerics, it is possible to implement solvers with optimal efficiency on dynamically adaptive (that is fastly changing) grids. Among others, this prevents a non-tolerable overhead for re-assembly of system matrices. Peano's k-spacetrees yield an especially tailored, plain, and simple type of grid. Such a fixed spatial discretisation together with a fixed traversal and the matrix-free solver concept restricts the algorithmic freedom, the feature flexibility, and the concrete realisation of any solver implemented within the framework. In particular, it inhibits the usage of out-of-the-box solvers for systems of linear equations and sophisticated external preconditioners. However, only with this restriction to matrix-free solvers and the tight connection between the grid, traversal algorithms, and solver numerics, it is possible to implement solvers with optimal efficiency on dynamically adaptive (that is fastly changing) grids. Among others, this prevents a non-tolerable overhead for re-assembly of system matrices and ensures low memory demands and an inherent multilevel grid hierarchy at all stages of the solver algorithm. Any plugin benefits directly from these properties. At least for CFD, the advantages outnumber any restrictions as this paper illustrates.

3 The Peano framework

The grid, its traversal, the data management, the domain decomposition, the load balancing, and so forth, are fixed in Peano. However, the application domain to be tackled as well as the spatial discretisation method on the grid, however, can be freely chosen. Section 4 presents a lower and a higher order discretisation for computational fluid dynamics. Such a PDE solver plugs into the traversal automaton's transitions, and, in this section, we describe this plugin mechanism, i.e. Peano's framework idea, that is the basis for any plugin.

Peano's architecture follows a rigourous encapsulation and information-hiding paradigm, and the code is a pure C++ implementation following first-of-all the composite and visitor pattern [11]. As a result, the framework is easy to use and extend, and any improvements of the grid traversal and management immediately carry over to any plugin of the framework. Due to a rigourous application of the flyweight pattern, the performance overhead induced by object-oriented techniques is reduced to a constant simulation startup penalty.

The user-defined plugins solving different (parts of the) PDE and the individual substeps of a PDE solver differ in the operations called during the grid traversal. In contrast to other frameworks (e.g. [4,19]), Peano implements an event-based approach instead of an iterator-based interface: The grid traversal and data management algorithms are fixed and Peano provides so-called traversal-events (see Table 1) such as *enterElement*, *touchVertexFirstTime*, and *touchVertexLast Time*. Users implementing a new solver or a new application in Peano can plugin their operations in these events.

Peano's optimised grid traversal is realised with a finite automaton. The traversal is similar to a depth-first search of the k-spacetree and provides the traversal events as a fixed set of call-back points, i.e. whenever the finite automaton performs a state transition, it gives the application the possibility to interact with the grid constituents. The plugins register for the call-back points (events), and, whenever an event is triggered, all registered plugin methods are invoked, i.e. one event can be mapped onto multiple components realising different features (solver iterations, dynamic refinement criteria, visualisation, and so forth). The implementations of these features are independent of each other. Furthermore, the shared- and distributed-memory parallelisation as well as the load balancing are hidden from the PDE implementation, as each subdomain invokes in parallel a set of events on its own. Finally, realising all the call-backs with a static polymorphism [31], the flexibility to exchange and combine arbitrary plugins does not introduce a performance breakdown.

A typical Peano application exhibits a three-layered architecture (Fig. 3): As foundation, Peano itself provides the adaptive Cartesian grid together with the traversal algorithm and several basic modules for data structures and communication (bottom layer). The grid is distributed among several computing nodes, and if it changes, the Peano core automatically redistributes the partitions. The grid component is a generic class structure parameterised over the application data. To realise an application, a user writes a runner steering the whole application, setting up and shutting down the computation, providing a user interface and so forth (top layer). This runner also selects and instantiates the PDE-specific plugins—the sandwich layer. Whenever the runner invokes the iterate operations on the grid component, the whole grid is traversed and the corresponding events are delegated to all the plugins chosen by the runner.

The event concept hides the grid management and the traversal details from the plugin developer. In accordance with this information hiding principle, Peano also hides the grid construction process and any grid modifications. As grid

Table 1 Subset of events triggered by the Peano framework

Event	Description
createDoF	Traversal creates cell or vertex with degree of freedom (DoF) due to the grid construction process or dynamic refinement. Application sets initial conditions, boundary information, and so forth
destroyDoF	Counterpart of *createDoF*. Is triggered by a grid coarsening
beginTraversal	Invoked whenever traversal starts
endTraversal	Invoked after traversal has terminated
enterElement	Traversal automaton enters a cell. Operation provides both the element, its spatial position, and the adjacent vertices
leaveElement	Traversal automaton leaves a cell
touchVertexFirstTime	Traversal automaton reads a vertex the first time throughout a traversal. Passes the vertex together with the spatial position
touchVertexLastTime	Traversal automaton has used a vertex for the last time throughout the traversal. It will not need this vertex afterwards
createTemporaryVertex	Create a hanging node. The hanging nodes are created on-the-fly, i.e. at least once per traversal
destroyTemporaryVertex	Counterpart of *createTemporaryVertex*

Plugins catch these events and modify the event arguments to implement PDE-specific behaviour

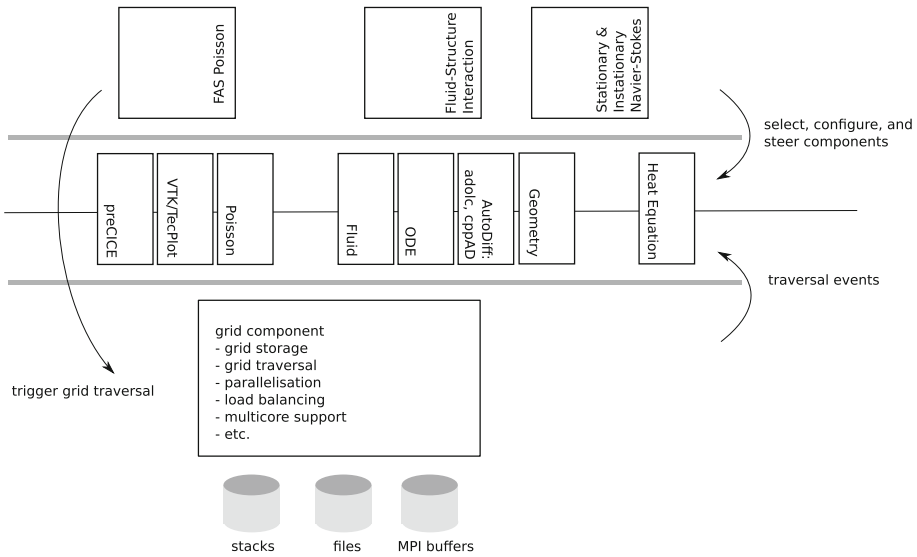

Fig. 3 Schematic view of the layers of the Peano software design. The layers below the fat black line (traversal events) are completely hidden from users implementing their application in Peano

construction and dynamical adaptivity are central functionalities in Peano, we describe the underlying algorithms in more detail: A (local) change of the refinement depth in the Peano grid is split up into two phases. Any plugin is allowed to invoke a refine or coarse operation on any vertex. Thereby, it indicates that all the cells that are adjacent to the vertex on the same level shall be refined or coarsened, respectively. Peano internally keeps book of all the refinement and coarsening wishes and executes them before the affected grid (parts) are used in the next iteration. That is, if a refinement is invoked, the actual refinement takes place one iteration later (Fig. 4). In-between the refinement/coarsening invocation and the actual change of the refinement depth, the corresponding events *createDoF* or *destroyDoF* are triggered and create new data or eliminate obsolete data before the event *touchVertexFirstTime* of the respective new or eliminated vertices are called. Consequently, two invariants hold for any vertex. On the one hand, each *touchVertex FirstTime* is followed by a *touchVertexLastTime* event. On the other hand, all the 2^d cells adjacent to a vertex have been traversed whenever the traversal triggers *touchVertex LastTime* for this vertex.

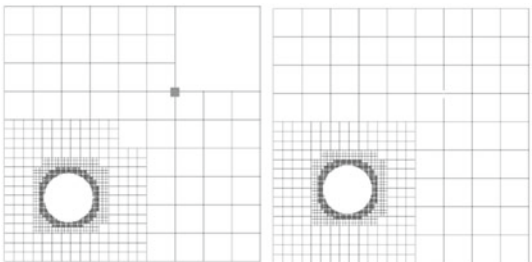

Fig. 4 A sphere is moving through a rectangular domain. If a refinement for a vertex is triggered (*left* illustration, *grey rectangular box*), all the 2^d adjacent cells on the respective level are refined in the iteration afterwards (*right* illustration)

4 Computational fluid dynamics in Peano

Within the discussion of the benefits arising from Peano, we study the simulation of incompressible laminar flows as discussed in [6]. Here, we give a more technical accent to our research work and present new features of the flow solver such as moving geometries in adaptively refined grids and higher order discretisation. The implementation is based on the Navier–Stokes equations discretised with tri-/bilinear finite elements or, alternatively, a higher order interpolated differential operator (IDO) approach [1,15,16]. For the time discretisation, different explicit schemes (forward Euler, fourth order Runge–Kutta) and implicit schemes (backward Euler (adaptive), trapezoidal rule) are provided.

As shown in Fig. 3, Peano's CFD application consists of several components implemented on top of the grid traversal events: The fluid component realises the evaluation of spatial operators via element matrices, the (staggered) Poisson component solves the occurring linear and nonlinear systems of equations with a built-in matrix-free multigrid solver. The ODE (ordinary differential equations) component finally provides the different time integrations schemes. All three components are realised and tested independently of each other and can be used for different PDE, too.

In the following, we describe the CFD realisation by means of a forward Euler time-step using the Chorin projection method. Applying a spatial discretisation of the Navier–Stokes equations with a diffusion matrix D_h, a convection tensor C_h, a divergence matrix M_h, and the discrete pressure gradient operator M_h^T, one time step contains the following

equations:

$$\mathbf{u}_h^{(n+1)} = \mathbf{u}_h^{(n)} + dt \underbrace{\left(\frac{1}{Re} D_h \mathbf{u}_h^{(n)} - \left(\mathbf{u}_h^{(n)}\right)^T C_h \mathbf{u}_h^{(n)}\right)}_{=: rhs^{(n)}}, \tag{1}$$

$$M_h^T M_h p_h = -\frac{1}{dt} M_h^T \mathbf{u}_h^{(n+1)}, \tag{2}$$

$$\mathbf{u}_h^{(n+1)} = \mathbf{u}_h^{(n+1)} - dt\ M_h p_h \tag{3}$$

with the discrete velocity field $\mathbf{u}_h^{(n)}$ and $\mathbf{u}_h^{(n+1)}$ at time $t^{(n)}$ and $t^{(n+1)}$, respectively, Reynolds number Re, and discrete pressure p_h. In the implementation, velocity values are stored in the vertices of the elements, whereas pressure values are stored in the element midpoints.

First, a preliminary velocity is computed in Eq. (1). Afterwards, a pressure Poisson equation (2) has to be solved and, last, the new, divergence-free velocity field is computed according to Eq. (3). Hereby, the fluid component takes care of the computation of the right-hand side $rhs^{(n)}$ in (1), the ODE component performs the time-step in (1), the Poisson (staggered) component solves (2) using an iterative solver (Gauss–Seidel, SOR, or geometric multiplicative multigrid), and the fluid component finally updates $\mathbf{u}_h^{(n+1)}$ according to (3).

Tables 2 and 3 give two examples how actions of the Navier–Stokes solver are mapped to the grid traversal events. Most of these mappings are straightforward at least for a second order discretisation, where the evaluation of the right-hand side of Eq. (1), e.g. for each vertex only involves data from directly neighbouring vertices (Fig. 5, left). Such a scheme fits to Peano's element-wise traversal. The splitting of the operators into cell-parts is trivial: each cell contributes a part that can be calculated using only the values at its vertices and adds it to the operator value. Such, the whole operator value is readily computed after all neighbouring cells of the respective vertex have been visited. For the evaluation of the pressure operator $M_h^T M_h p_h$ in (2), the splitting into cell-parts is not so obvious but still can be done with the same efficiency: The pressure Laplacian in a second order discretisation requires pressure values stored at the midpoints of neighbouring elements (see Fig. 5, middle and right). As these values are not directly available, we make them accessible indirectly by storing pressure gradients $\mathbf{grad}_h := M_h p_h$ at the element vertices. Thus, the residual of Eq. (2) can be calculated locally within an element as

$$res_p = -\frac{1}{dt} M_h^T \mathbf{u}_h^{(n+1)} - M_h^T \mathbf{grad}_h. \tag{4}$$

The residual is calculated in the event *enterElement* and immediately used both to update the pressure value and the pressure gradients at the element vertices using the element-parts of the matrix M_h for all adjacent vertices. Thus, no additional grid traversal is required. In a more sophisticated version, even the storage of the pressure gradient $\mathbf{grad}_h$ can be omitted as the residual (4) can also be computed from

$$res_p = -\frac{1}{dt} M_h^T \mathbf{u}_h^{(n+1)} \tag{5}$$

if we immediately update the velocities $\mathbf{u}_h^{(n+1)}$ according to

$$\mathbf{u}_h^{(n+1)} = \mathbf{u}_h^{(n+1)} - dt\ M_h res_p. \tag{6}$$

This is again done using element-parts of the matrix M_h. Such, neither an additional grid traversal nor additional storage are required to solve the pressure Poisson equation (2) that, at first sight, possesses an operator that contradicts the element-wise traversal strategy of Peano. This single level scheme can be extended to a matrix-free multigrid solver also working in a cell-wise manner. For the interlevel operations restriction and interpolation, data of a cell and the corresponding son cells but not of neighbouring cells are required. An additional difference to single level solvers is that not the whole spacetree but only the tree parts up to the current grid level are required within a multiplicative multigrid solver. For this purpose, Peano implements a handling mechanism for multigrid coarsening and refinement that works almost the same as the coarsening and refinement for dynamical grid adaptivity with the only difference that data are not thrown away completely during coarsening but written to suitable auxiliary data structures and not newly created during refinement but read from these auxiliary data structures. With a careful design of these auxiliary data structures as data stacks, these operations do not lower the hardware efficiency of the multigrid solver compared to a single level solver such as Gauss–Seidel.

The evaluation of higher order discretisations can be realised with Peano's cell-oriented approach as shown in [23]. Similar to the pressure Laplacian, auxiliary values storing contributions of neighbouring values that are not directly accessible are used. If we look at the example of the interpolated differential operator approach (IDO) [1], face and cell integrated velocity data have to be stored in addition to the vertex values (Fig. 6, left). As Peano technically only allows to store data at vertices or cell midpoints, face data are stored in the respective left or lower grid vertex in the two-dimensional case, face and edge data in the three-dimensional case have to be stored in vertices in a similar way. For the evaluation of the higher order discretisation version of the right-hand side in Eq. (1), However, not only such vertex, face, edge, and cell data of one cell are required but also data of neighbouring cells. Here, data of the respective neighbours written to the current cell's vertices in a first step to allow for the operator accumulation in a second step (Fig. 6, middle and right). Within a linear solver, the second step again can be intertwined with the updating of degrees of freedom and the respective transported neighbour values in one grid traversal analogue to the case of the pressure Poisson equation.

Table 2 Actions taken in traversal events during the calculation of the right-hand side in Eq. (1)

Traversal event	Action
createDoF	Create new velocity and pressure data by interpolation from existing vertex data
destroyDoF	Delete velocity and pressure data
beginTraversal	–
endTraversal	–
enterElement	Evaluate cell-parts of $D_h\mathbf{u}_h^{(n)}$ and $\mathbf{u}_h^{(n)}C_h^T\mathbf{u}_h^{(n)}$ and add it to the accumulated values of the right-hand side
leaveElement	–
touchVertexFirstTime	Reset accumulated value of the right-hand side to zero
touchVertexLastTime	–
createTemporaryVertex	Calculate interpolated velocity values from vertices of the respective father cell
destroyTemporaryVertex	Restrict cell-parts of operator values to vertices of the respective father cell, eliminate interpolated velocity values

Table 3 Actions taken in traversal events during one (Gauss–Seidel) smoother iteration of the solver for the pressure Poisson Eq. (2)

Traversal event	Action
createDoF	–
destroyDoF	–
beginTraversal	Reset the residual norm to zero
endTraversal	Finish the calculation of the residual norm
enterElement	Calculate the local residual and immediately update the pressure value at the element midpoint and the pressure gradients at the element vertices
leaveElement	–
touchVertexFirstTime	–
touchVertexLastTime)	–
createTemporaryVertex	Interpolate pressure gradients from pressure gradients at the vertices of the respective father cell
destroyTemporaryVertex	Restrict gradient updates to the gradients at the vertices of the respective father cell

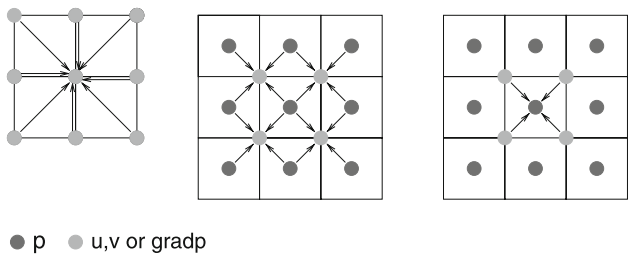

Fig. 5 Values required for the evaluation of the right-hand side of (1) at a vertex (*left*); values required for the evaluation of the pressure Laplacian in (2) in the midpoint of a cell (*middle* and *right*). The modified cell-wise evaluation of the pressure Laplacian in (2) first computes pressure gradients at the cells vertices (*middle*) and then computes the pressure Laplacian from the gradients (*right*). In a solver, both parts are intertwined in one grid traversal: each time, a pressure value is updated, for example during a Gauss–Seidel solver iteration, immediately the values of the gradients at all cell vertices are updated as well. Optimised solver versions purely work with pressure values at midpoints and velocities at vertices

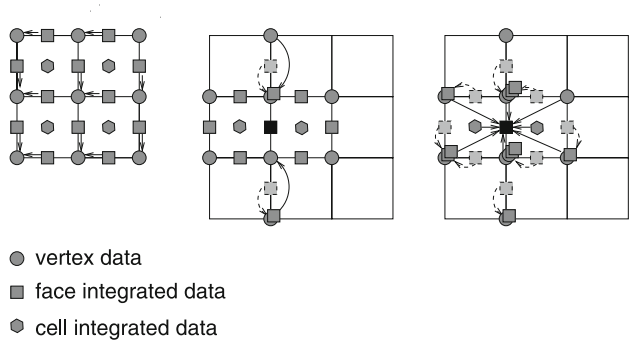

Fig. 6 Usage of auxiliary values for the evaluation of the diffusion coefficient in (1) following the higher order IDO [1]

5 Numerical results

This section presents different results of simulations carried out with Peano. We first focus on performance aspects such as memory efficiency, parallel speedup, and multigrid performance. Afterwards, different flow scenarios are presented that show the correctness of the implementation and the support for complex and moving geometries of Peano's CFD solver. All experiments were conducted on standard 32 bit Intel Pentium dual core processing units with two MB L2 cache size, and on the processing units of the HLRB II (Höchstleistungsrechner Bayern II) at the Leibniz Supercomputing Centre in Munich—an SGI Altix 4700 with 64 bit Itanium2 Montecito dual cores with 256 KB L2 and 9MB L3 caches.

5.1 Performance results

In this section, we present some general performance results concerning both hardware and numerical efficiency. As mentioned in Sect. 2, a great advantage of spacetree grids in combination with our data structures and data access algorithms is the high memory efficiency. For CFD simulations with Peano, Table 4 shows the necessary memory in bytes for cells and vertices in two and three dimensions. Depending on the time integration scheme chosen at runtime (forward Euler, fourth-order Runge–Kutta, or adaptive trapezoidal rule), not more than 200 bytes are consumed for a grid vertex including values of degree of freedom (velocities) and all grid data. This is substantially less than several thousand bytes that typically are necessary in comparable PDE frameworks based on flexible grid types or unstructured grids and explicit matrix assembly.

Note that due to the efficient combination of spacetrees, stack data structures, and the space-filling Peano curve all simulations with Peano show a very high cache hit-rate above 98% independent of the dimension d, the adaptivity pattern of the grid, and the type of solver and PDE in use.

The grid traversal component runs in parallel and, thus, provides parallelisation support for any application. Figure 7 shows the (weak) speedup of a linear solver running on the HLRB II with up to 900 processors. Each partition contains approximately 6×10^4 ($d = 2$) or 7×10^5 ($d = 3$) vertices, respectively. Note that not only solver iterations but also the domain decomposition itself run in parallel. The breakdown of the speedup at certain numbers of processors is induced by the partitioning strategy and improved in currently ongoing work.

To validate the multigrid performance of our pressure Poisson solver executed in each time step, we solved a pressure Poisson equation with an artificial right-hand side for which the exact solution is known. Figure 8 shows the corresponding resolution-independent convergence speed of the multiplicative V-cycle.

5.2 Applications

This section presents results of CFD applications computed with Peano. Benchmark simulations allow to compare the results with hard reference data to judge the benefit of adaptivity. Furthermore, we present flow simulations using moving and more complex geometries in combination with Peano's spacetree grids.

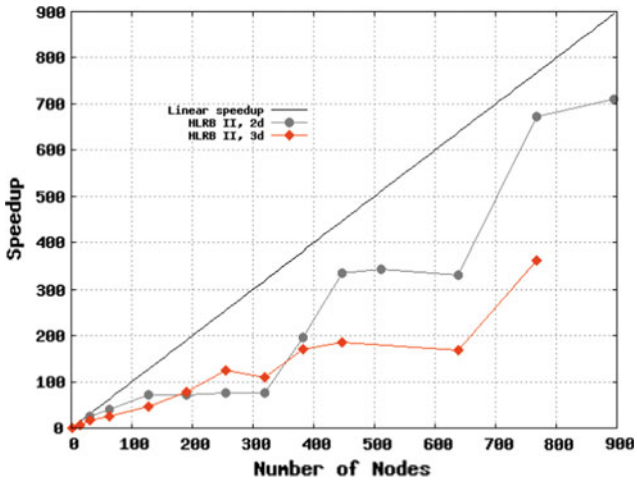

Fig. 7 Weak speedup of a matrix-free linear solver on the HLRB II at the Leibniz-Rechenzentrum in Munich (results taken from [32])

5.2.1 Benchmark flow around a cylinder

The well-known "Benchmark computations of laminar flow around a cylinder" [30] consist of several two- and three-dimensional scenarios with an obstacle located near the entry of a channel. Besides other data, the drag and lift coefficients of the force exerted by the flow onto the cylinder serve as reference values to compare the accuracy of results. Figure 9 visualises the adaptive Cartesian grid and the horizontal velocity solution of the steady-state setup 2D–1 at Reynolds number $Re = 20$.

For a suite of simulations of 2D–1 on different adaptive grids, Table 5 shows the minimum and maximum level of refinement of the underlying spacetree, the corresponding number of degrees of freedom (DoF), the drag and lift coefficients, and the CPU time in seconds per time step are indicated. The keyword "box" refers to manual refinement boxes used to identify regions of refinement (as in Fig. 9) instead of refinement due to the mere geometry boundaries.

Table 4 Memory requirement of the Navier–Stokes solver in Peano with different time integration schemes on adaptive grids in Peano

Dimension	Cell in bytes		Vertex in bytes		
	Default	TR adap.	FE	RK	TR adap.
$d = 2$	40	48	92	124	144
$d = 3$	44	52	124	172	200

As a spatial discretisation, a second order finite element method is used. The values are indicated in bytes

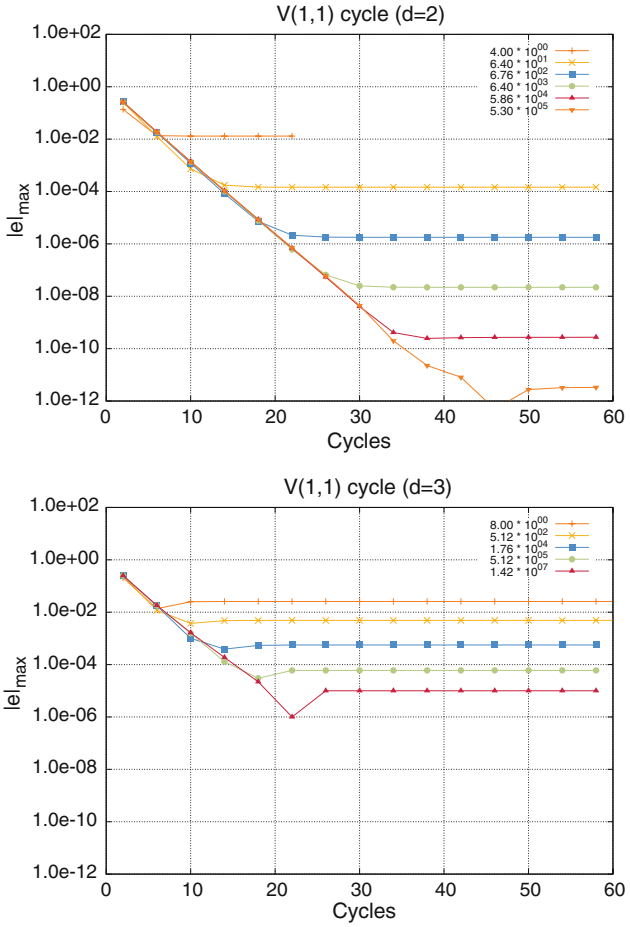

Fig. 8 V-cycle behaviour of the PPE solver for a stationary problem on a regular grid. The analytical solution of the continuous problem is known. After a few v-cycles, the error (measured in the $|.|_{max}$ norm) converges to the discretisation error. The convergence speed is independent from the grid resolution

The first two rows of Table 5 demonstrate exemplarily the expected benefit of adaptivity: A significantly lower number of unknowns, resulting in a shorter runtime, is sufficient to obtain nearly the same results as in the regular case (indicated by identical maximum and minimum refinement level of eight). Investing in more DoF near to the obstacle results in better approximations of the reference data as the remaining three rows of Table 5 clearly show. Besides, the runtimes behave quasi-linearly and, thus, are independent of the balance of the spacetree, i.e. Peano does not pay a higher price for a very steep adaptivity compared to a more regular grid.

The evaluation of the benefit of dynamical adaptivity using an automatic refinement criterion instead of the geometry boundaries or manual boxes is subject to current work. This and an enhanced boundary treatment at the circle surface, for example by a boundary-adapted grid refinement, will increase the accuracy of the solver that obviously is not second order yet in Table 5.

Comparisons with Peano's straightforward regular grid implementation and other solvers showed that the overhead of adaptivity is about 30–80% (decreasing with increasing problem size) in 2D and drops to only about 3% for three-dimensional scenarios (see [23]).

5.2.2 *Moving geometries*

To demonstrate the correct implementation of moving geometries in Peano's CFD solver, we use a solid sphere moving with a constant velocity from right to left in a box that possesses an outlet on the right-hand side. Figure 10 shows qualitative results of this scenario with a coarse grid resolution: The adaptive Cartesian grid is changing (i.e. refining and coarsening) near the geometry boundaries, and the streamlines in grey indicate the changing flow behaviour. A comparison of the forces on the sphere with results of an equivalent scenario with a fixed sphere and constant flow from the left-hand side (see [23]) showed good accordance of the reference values. This example shows the great potential of Peano: The dynamical grid adaptivity is very cheap. Refinements and coarsenings fit seamlessly into the cache-optimised data storage and data access concept of Peano and require only one single grid traversal, that, in addition can include at the same time the next solver computations. This allows for a very fast realisation of even large geometry or even topology changes within a fixed grid approach compared to Lagrangian or arbitrary Lagrangian–Eulerian (ALE) approaches.

Table 5 Survey of adaptive simulation results of the benchmark 2D–1 [30] for different maximum and minimum tree levels

Max. level	Min. level	# DoF	c_d	c_l	CPU time per time step
8	8	1,051,253	5.682	0.0146	9.35 s
8	6 (box)	88,857	5.680	0.0150	0.71 s
9	7	125,041	5.591	0.0113	1.03 s
9	8	10,57,877	5.561	0.0112	9.46 s
9	6 (box)	261,501	5.586	0.0115	2.46 s
ref. data		–	5.580	0.0107	–

The drag (c_d) and lift (c_l) coefficients of the cylinder and the runtime of one time step in seconds are shown in the last three columns. The reference data from [30] are given in the last line

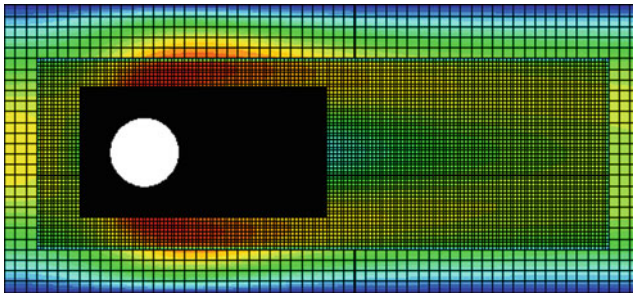

Fig. 9 Horizontal velocity distribution and adaptive Cartesian grid of the benchmark 2D–1 [30] at $Re = 20$ with maximum and minimum tree level nine and six, respectively

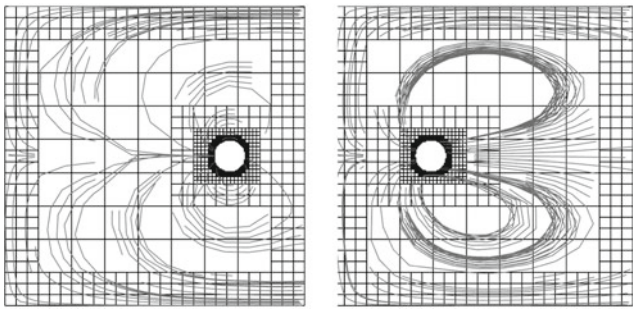

Fig. 10 Snapshots of a box scenario with outlet on the right-hand side containing a sphere that moves from right to left (see streamlines in *grey*). The adaptive Cartesian grid (*black*) is refined and coarsened accordingly

5.2.3 Complex geometries

Peano is able to handle more complex geometries. One example is the drift ratchet scenario that uses sinusoidal, asymmetric pore geometries which are implicitly defined (see [7,20] for details on the scenario and preliminary results). Here, we performed a two-dimensional simulation on four subsequent pores using time-dependent pressure boundary conditions resulting in an oscillating flow scenario. Figure 11 shows the channel-like pressure distribution in the four pores as well as a snapshot of the curvilinear geometry of a single pore with the resulting adaptive grid. The maximum level of the underlying spacetree at the boundary of the geometry is nine whereas a minimum level of six is used in the rest of the domain. This results in 98,521 cells, 82,144 vertices and 33,948 hanging nodes.

6 Conclusion and outlook

The aim of this paper is twofold: On the one hand, it introduces the Peano framework. On the other hand, it shows that the adaptive Cartesian grids underlying the framework are well-suited for state-of-the-art CFD applications. It thus

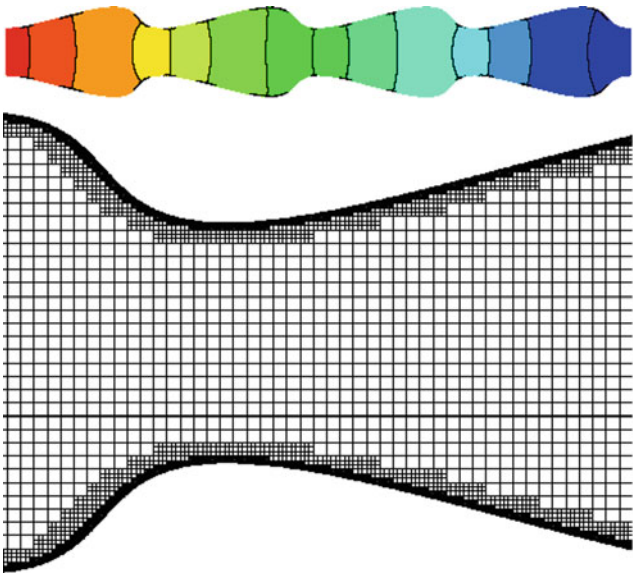

Fig. 11 Pressure distribution (*top*) and snapshot of the corresponding adaptive grid (*bottom*) of a 2D drift ratchet scenario (cf. [7,20]) with maximum and minimum spacetree refinement level of nine and six, respectively

bridges the gap from a pure software presentation to a real world application scenario.

Peano combines, in a novel way, modern multiscale algorithms for higher-order methods running on dynamically adaptive grids with a very low memory footprint. It runs on massive parallel multicore clusters and efficiently exploits the memory hierarchy of today's computers. Due to this efficiency, we can afford to refine the grid flexibly, and we show that computational fluid dynamics on such adaptive Cartesian grids are able to cope with unstructured adaptive grid approaches in terms of accuracy. At the same time, Peano fully exploits all advantages of well-structured grids in an optimised implementation.

Besides the fact that we meanwhile use Peano for applications different from fluid dynamics, we next use Peano's CFD plugin to tackle challenges that were out of scope before. On the one hand, these are challenges such as turbulence research via direct numerical simulation where the experimental setup requires an extremely fine grid. Here, Peano's low memory footprint proves of value whereas other codes cannot resolve the problem spatially with sufficient accuracy. On the other hand, these are classical fluid-structure interaction challenges where the grid structure and, consequently, the load balancing, and the global system matrix have to change permanently. Here, Peano's on-the-fly adaptivity, its low memory overhead, and its matrix-free philosophy prove of value whereas other codes depend on an expensive re-meshing, re-assembling, and re-balancing from time to time. While Peano already laid the foundations of some new scientific

insights (e.g. [6,7]), the overall potential is, from our point of view, far from exhausted, will be subject of further studies, and will be starting point for methodological extensions.

Methodologically, Peano offers at least three additional charming chances for CFD codes: First, the current CFD implementation does not exploit a dimension $d \geq 4$. However, the d-dimensional concept of Peano allows for a straightforward enhancement to higher-dimensional grids for of an explicit resolution of the time (together with space–time adaptivity), parameter studies, and optimisation scenarios. Second, the current implementation does not offer p-adaptivity, yet. The integration of p-adaptivity will facilitate an efficient usage of today's massive parallel vector arithmetic units, i.e. yield impressing MFlop rates, while improving the solution's accuracy, too. Finally, the current framework is a stand-alone solution. To make it a tool used by many scientists and engineers to produce new insight, it has to be integrated into today's scientific computing landscapes. To open Peano's signature to the public is the first step towards such an integration. In return, Peano has to support, connect, and integrate to de facto standard interfaces such as standard geometry formats, postprocessing and visualisation toolkits, as well as computational steering environments. First steps in this direction have already been made by defining a triangulation-based geometry interface and clearly defined output interfaces.

Download

Peano is open-source software and available at www5.in.tum.de/peano under a BSD-like license.

Acknowledgments Thanks are due to Christoph Zenger for a lot of valuable input and fruitful discussions. Kristof Unterweger implemented the moving geometry feature within the Peano framework throughout his master's thesis. The ongoing financial support of TUM's International Graduate School of Science and Engineering (IGSSE) is gratefully acknowledged.

References

1. Aoki T (1997) Interpolated differential operator (IDO) scheme for solving partial differential equations. Comput Phys Comm 102:132–146
2. Bader M, Bungartz H-J, Frank A, Mundani R-P (2002) Space tree structures for PDE software. In: Proceedings of the International Conference on Computer Science (3), vol 2331, p 662
3. Bader M, Frank A, Zenger C (2002) An octree-based approach for fast elliptic solvers. High Perform Sci Eng Comput 21:157–166
4. Bastian P, Blatt M, Dedner A, Engwer C, Klöfkorn R, Ohlberger M, Sander O (2008) A generic grid interface for parallel and adaptive scientific computing. Part I: Abstract Framework. Computing 82(2–3):103–119
5. Borrmann A, Schraufstetter S, Rank E (2009) Implementing metric operators of a spatial query language for 3d building models: octree and b-rep approaches. J Comput Civil Eng 23(1):34–46
6. Brenk M, Bungartz H-J, Daubner K, Mehl M, Muntean IL, Neckel T (2008) An Eulerian approach for partitioned fluid-structure simulations on Cartesian grids. Comput Mech (accepted)
7. Brenk M, Bungartz H-J, Mehl M, Muntean IL, Neckel T, Weinzierl T (2008) Numerical simulation of particle transport in a drift ratchet. SIAM J Sci Comput 30(6):2777–2798
8. Deering M (1995) Geometry compression. In: SIGGRAPH '95: proceedings of the 22nd annual conference on computer graphics and interactive techniques. ACM Press, New York, pp 13–20
9. Düster A, Bröker H, Heidkamp H, Heißerer U, Kollmannsberger S, Krause R, Muthler A, Niggl A, Nübel V, Rücker M, Scholz D (2004) AdhoC^4—user's guide. Lehrstuhl für Bauinformatik, Technische Universität München
10. Fuster D, Baguéa A, Boeckc T, Le Moynea L, Leboissetierd A, Popinete S, Raya P, Scardovellif R, Zaleskia S (2009) Simulation of primary atomization with an octree adaptive mesh refinement and vof method. Int J Multiph Flow 35(6):550–565
11. Gamma E, Helm R, Johnson RE, Vlissides J (1994) Design patterns—elements of reusable object-oriented software, 1st edn. Addison-Wesley, Longman
12. Gao F, Ingram DM, Causon DM, Mingham CG (2007) The development of a Cartesian cut cell method for incompressible viscous flows. Int J Numer Meth Fluids 64(9):1033–1053
13. Gerstenbrger A, Wall WA (2007) An extended finite element method/mortar method based approach for fluid-structure interactions. Comput Methods Appl Mech Eng 197:1699–1714
14. Harlow FH, Welch JE (1965) Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface. Phys Fluids 8(12):2182–2189
15. Imai Y, Aoki T (2006) A higher-order implicit IDO scheme and its CFD application to local mesh refinement method. Comput Mech 38:211–221
16. Imai Y, Aoki T, Takizawa K (2008) Conservative form of interpolated differential operator scheme for compressible and incompressible fluid dynamics. J Comput Phys 227:2263–2285
17. Klass O, Shephard MS (2000) Automatic generation of octree-based three-dimensional discretisations for partition of unity methods. J Comput Mech 25(2–3):296–304
18. Lam TW, Yu KM, Cheung KM, Li CL (1998) Octree reinforced thin shell objects rapid prototyping by fused deposition modelling. Int J Adv Manufact Technol 14(9):631–636
19. Long K (2009) Sundance: a rapid prototyping toolkit for parallel pde simulation and optimization. In: Heinkenschloss M, Biegler LT, Ghattas O, van Bloemen Waanders B (eds) Large-scale PDE-constrained optimization. Lecture notes in computational science and engineering, vol 30. Springer, Berlin, pp 331–339
20. Matthias S, Müller F (2003) Asymmetric pores in a silicon membrane acting as massively parallel brownian ratchets. Lett Nat 424:53–57
21. Mittal R, Iaccarino G (2005) Immersed boundary methods. Annu Rev Fluid Mech 37:239–261
22. Morton GM (1966) A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Ontario
23. Neckel T (2009) The PDE framework Peano: an environment for efficient flow simulations. Verlag Dr. Hut, München
24. Sagan H (1994) Space-filling curves. Springer, New York
25. Samet H (1984) The quadtree and related hierarchical data structures. ACM Comput Surv 16(2):187–260
26. Sampath RS, Adavani SS, Sundar H, Lashuk I, Biros G (2008) Dendro: parallel algorithms for multigrid and amr methods on 2:1 balanced octrees. In: SC '08: proceedings of the 2008 ACM/IEEE conference on supercomputing, Piscataway, NJ, USA. IEEE Press, New York, pp 1–12

27. Sundar H, Sampath RS, Biros G (2008) Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. SIAM J Sci Comput 30(5):2675–2708
28. Tomé MF, McKee S (1994) GENSMAC: a computational marker and cell method for free surface flows in general domains. J Comput Phys 110:171–186
29. Tu T, O'Hallaron DR, Ghattas O (2005) Scalable parallel octree meshing for terascale applications. In: SC '05: proceedings of the 2005 ACM/IEEE conference on supercomputing, Washington, DC, USA. IEEE Computer Society, New York, p 4
30. Turek S, Schäfer M (1996) Benchmark computations of laminar flow around a cylinder. In: Hirschel EH (ed) Flow simulation with high-performance computers II, NNFM, vol 52. Vieweg, Braunschweig
31. Vandevoorde D, Josuttis N (2003) C++ templates—the complete guide. Addison-Wesley, Reading
32. Weinzierl T (2009) A framework for parallel PDE solvers on multiscale adaptive Cartesian grids. Verlag Dr. Hut, München