TUM

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

PEANOCLAWâ• A FUNCTIONALLY-DECOMPOSED APPROACH TO ADAPTIVE MESH REFINEMENT WITH LOCAL TIME

Kristof Unterweger, Tobias Weinzierl, David I. Ketcheson, Aron Ahmadia

TUM-I1332

Technischer Bericht
Technische Universität München
Institut für Informatik

PEANOCLAW—A FUNCTIONALLY-DECOMPOSED APPROACH TO ADAPTIVE MESH REFINEMENT WITH LOCAL TIME STEPPING FOR HYPERBOLIC CONSERVATION LAW SOLVERS

KRISTOF UNTERWEGER *, TOBIAS WEINZIERL †, DAVID I. KETCHESON ‡, AND ARON AHMADIA §

Abstract. We present an application framework applying spacetree-based adaptive mesh refinement (AMR) to solvers for hyperbolic partial differential equations (PDEs) specified on logically quadrilateral grids. The AMR framework decomposes the adaptive grid into regular quadrilateral subgrids shaping an adaptive global grid, traverses these subgrids autonomously, calls PDE-specific routines on each subgrid, and preserves the data consistency between the subgrids. Each subgrid is autonomously allowed to advance in time based on the local CFL condition, yielding a speedup in several cases compared to global time stepping. The AMR memory footprint is small due to the use of nonoverlapping grids. Subroutines written for regular Cartesian grids are used on adaptive meshes without modification. Furthermore, the framework provides a very simple programming interface to specify dynamic refinement criteria. We thus lower the implementation threshold for domain specialists who want to extend existing code with AMR features without introducing complexity into their own applications. Our framework is a merger of the spacetree mesh management and traversal code Peano and Clawpack's PyClaw, an explicit finite volume solver for general PDEs.

1. Introduction. Adaptive mesh refinement (AMR) is a technique for solving hyperbolic partial differential equations (PDEs) whose solution and regions of interest vary significantly in space. AMR hierarchically applies grids with a finer resolution where the solution changes rapidly, is non-smooth, or, in general, exhibits behavior insufficiently resolved by coarse grids. Since the grid resolution determines the computational workload and memory footprint, AMR makes it possible to spend computing time and memory where they pay off most. AMR is a strongly desired feature for simulation codes in many application fields when using a uniform grid that is fine enough to resolve localized fine structures is not an option due to memory and runtime restrictions. Additionally, a well-designed AMR implementation allows coarse grids to avoid the time step restrictions imposed by the CFL condition on highly resolved grids [5, 6, 7, 12, 19].

Although AMR is of great use in certain application domains, many simulations that could benefit from AMR still are performed at uniform resolution. We discuss three possible reasons for this. First and foremost, implementation of adaptive mesh refinement is a complex software engineering task, but the principal authors of numerical software are typically not software engineers. This also means that implementation of better models or discretizations often takes priority over improvements like AMR. Second, high performance numerical kernels (here we use the term kernel to denote a function or algorithm acting on a simple piece/array of data) are tuned against simple, non-adaptive data structures such as Cartesian grids. Although this approach has demonstrated value in obtaining high single-core performance [4, 14], it is not immediately obvious how to adapt tuned kernels to dynamically adaptive data structures. Adapting such kernels to work with AMR is laborious and error-prone. Third, AMR adds CPU and memory overhead over Cartesian grid approaches. It

*Technische Universität München, 85748 Garching, Germany, {unterweg,weinzier}@in.tum.de
†Corresponding author
‡4700 King Abdullah University of Science and Technology, Thuwal 23955-6900, Kingdom of Saudi Arabia, david.ketcheson@kaust.edu.sa
§Columbia University, 1255 Amsterdam Avenue, New York, 10027, United States of America, aron@ahmadia.net

is not uncommon for adaptively refined grid implementations to have a significantly poorer data throughput than their uniform grid counterparts.

We address these challenges by coupling existing solvers for hyperbolic PDEs on regular Cartesian grids to tree-based AMR approaches. As proof-of-concept we introduce PeanoClaw, which combines the AMR PDE framework Peano [26] and PyClaw [20], a Python implementation of Clawpack. PyClaw comprises a collection of solvers (including those of Clawpack [18]) for hyperbolic PDEs on general quadrilateral grids [17]. The present work provides AMR to a hyperbolic solver as a black-box. In particular, the application developer does not implement the routines that maintain the mesh consistency, (i.e. invoke inter-grid mappings and ghost layer initialization) herself. Peano's spacetree and traversal are hidden from the user. Section 2 illustrates for a model problem how an existing PyClaw implementation benefits from PeanoClaw's AMR features with only minimal changes.

Peano's approach to AMR is based on the notion of a spacetree. Peano's spacetree formalism, algorithm design, and AMR programming interface are explained in Sections 3 and 6. Compared to the patch-based AMR of [5], one advantage of the present approach is that a given grid needs not cover the next finer grid completely. Instead, we propose an on-the-fly synchronization at grid resolution interfaces to preserve conservation. If necessary, our approach can be used without a coarsening scheme for non-ghost cells, since it does not maintain coarser grid representations in refined regions. Although it is costly, this can be useful, e.g., for shallow water equations near shorelines, where mass conservation is difficult to maintain during coarsening [19]—in that case, however, grid regions, once refined, would remain refined through the rest of the simulation. Compared to alternative spacetree approaches such as [2], the combination of Peano and PyClaw uses the spacetree directly as grid container and ties the grid traversal to the spacetree traversal. In particular, the grid traversal order and order of kernel invocations are subject of the tree code and not steered from outside—the tree code is not used as library but as framework into which the hyperbolic codes embed into. For a more complete discussion of Peano, see [25, 27]. Other approaches such as such as non-aligned grids, unstructured or triangular spacetree meshes [1] are beyond the scope of this work.

In Section 4 we describe PeanoClaw's local time stepping capability [15], which allows different subgrids to advance using independent time steps. Our aim is two-fold: we validate that the proposed data structures and control flows fit to conservative time stepping [11, 19], and we improve classical time stepping for multiple computational grids. For the latter, we augment and alter the AMR time stepping from [5] with the feature that subgrids are allowed to progress in time with different speed—similar to local adaptive time stepping methods on topologically flat unstructured grids [10, 12]. The decomposition of the time stepping among subgrids endows the PyClaw time stepping kernels with a multiresolution, multifront time stepping feature. Subgrids with a low information propagation speed can be updated less frequently than other grid regions, reducing computing time. Traditional time stepping is replaced by an iterative scheme advancing multiple subgrids per grid sweep in time. The overall simulation thus spans a time interval that monotonically advances towards a given termination time [11].

In Section 5, we present a rigorous formalism and a simple performance and memory model both for the local time stepping and the adaptive grids. Numerical experiments, described in Section 7 reveal how model problems benefit from our methodology. A summary and a brief outlook on future work comprise Section 8.

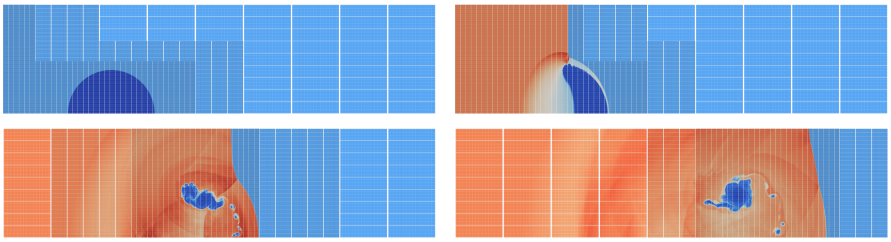


Fig. 2.1: Four snapshots of the shock-bubble interaction benchmark. A shock front enters the domain from the left and hits a bubble of different material (first row). The bubble material then is carried with the front and distorted (second row). Density and streakline markers for the bubble material are plotted.

All software discussed in this work is available for free under a permissible open source license [20, 26].

2. Example. As demonstrator use case, we consider an ideal gas with $\gamma = 1.4$ in the cylindrical domain $[0, 2] \times [0, 0.5]$. The Euler equations for a compressible, inviscid fluid with cylindrical symmetry can be written as

$$\begin{aligned} \rho_t + (\rho u)_z + (\rho v)_r &= -\frac{\rho v}{r}, \\ (\rho u)_t + (\rho u^2 + p)_z + (\rho uv)_r &= -\frac{\rho uv}{r}, \\ (\rho v)_t + (\rho uv)_z + (\rho v^2 + p)_r &= -\frac{\rho v^2}{r}, \quad \text{and} \\ (\rho E)_t + ((\rho E + p)u)_z + ((\rho E + p)v)_r &= -\frac{(\rho E + p)v}{r}. \end{aligned} \tag{2.1}$$

Here the z-coordinate represents distance along the axis of symmetry while the r-coordinate measures distance away from the axis of symmetry. The quantities ρ, E, p represent density, total energy, and pressure, respectively, while u and v are the z- and r-components of velocity.

The problem, whose solution is depicted in Figure 2.1, consists of a planar shock traveling in the z-direction that impacts a spherical bubble of lower-density fluid. In front of the shock $u = v = 0$ and $\rho = p = 1$ except inside the bubble, where $p = 1, \rho = 0.1$. Behind the shock, $p = 5, \rho \approx 2.82, v \approx 1.61$, and these conditions are also imposed at the left edge of the domain. Reflecting boundary conditions are imposed at the bottom of the domain while outflow conditions are imposed at the top and right boundaries.

The pure PyClaw code for this example is shown in Listing 1. The functions that set the initial conditions, compute the source terms, and set the left boundary condition are omitted for brevity. To make use of PeanoClaw and its adaptive mesh refinement, the code has to be modified as follows. First, the script has to use PeanoClaw's AMR solver wrapping the `ClawSolver2D` instance. For this, Line 35 and 36 have to be replaced by the code snippet of Listing 2. Second, the script has to control the adaptivity. For this, the additional operation `refinement_criterion` has to be defined (Listing 3). PeanoClaw decomposes the grid being regular so far into subgrids. For each subgrid it invokes `refinement_criterion` once per grid traversal with a state

Listing 1: Solution of the Shock-Bubble interaction problem with PyClaw

```
"""Solve the Euler equations of compressible fluid dynamics.
This example involves a bubble of dense gas that is impacted
by a shock."""

from clawpack import pyclaw
from clawpack import riemann
import numpy as np

gamma = 1.4

solver = pyclaw.ClawSolver2D(riemann.rp2_euler_5wave)
solver.num_waves = 5; num_aux=1
domain = pyclaw.Domain([0.,0.],[2.0,0.5],[160,40])
state  = pyclaw.State(domain,solver.num_eqn,num_aux)

state.problem_data['gamma']= gamma
state.problem_data['gamma1']= gamma - 1.

qinit(state)
auxinit(state)

solver.source_split = 1
solver.step_source=step_Euler_radial

solver.bc_lower[0]=pyclaw.BC.custom
solver.user_bc_lower=shockbc
solver.bc_lower[1]=pyclaw.BC.wall
solver.bc_upper[:]=pyclaw.BC.extrap

solver.aux_bc_lower[:]=pyclaw.BC.extrap
solver.aux_bc_upper[:]=pyclaw.BC.extrap

claw = pyclaw.Controller()
claw.tfinal = 0.2
claw.solution = pyclaw.Solution(state,domain)
claw.solver = solver

status = claw.run()
```

Listing 2: Changes for injecting the PeanoClaw solver into PyClaw control flow

```
claw.solver = pyclaw.peanoclaw.Solver(solver
, initial_mesh_width
, qinit
, refinement_criterion # cf. Listing 3
)
claw.solution = pyclaw.peanoclaw.Solution(state, domain)
```

Listing 3: Refinement criterion for PeanoClaw

```
def refinement_criterion(subgridstate):
  # subgridstate holds current mesh width, which is copied to
  # new_mesh_width_for_subgrid
  ...
  # Example refinement criterion:
  # if max_of_derivative_in_subgrid > refine_threshold:
  #   new_mesh_width_for_subgrid = new_mesh_width_for_subgrid / 4;
  # if max_of_derivative_in_subgrid < coarsen_threshold:
  #   new_mesh_width_for_subgrid = new_mesh_width_for_subgrid * 2;
  ...
  return new_mesh_width_for_subgrid
```

argument holding the solution, the grid resolution, as well as further meta data. The method now can alter the resolution and, this way, instruct PeanoClaw to remesh. Sophisticated error estimators do exist, but often a simple evaluation of the maximum of the absolute value of the solution gradient is sufficient (as comment in Listing 3): if it exceeds a given threshold, the grid is refined, if it underruns a given threshold, the grid can be coarsened. If the resolution remains unaltered, the grid is not changed. Many default values (subgrid sizes, interpolation routines, e.g.) do exist and can be modified.

3. Functional Decomposition. Although many numerical software packages are modular in nature, up until now, relatively few have provided an abstraction cleanly decoupling the choice of the underlying data storage format from the numerical algorithms operating on the data. This lack of separation is largely historical in nature, as programming languages and compilers were unable to provide good performance with a layer of function pointers or other indirection between functionality and data layout. Changing the data layout or decoupling functionality from a strongly coupled numerical software package that assumes a specific layout is *invasive*. By invasive, we mean that a substantial number of lines of code need to be changed. We propose a strategy that allows the application scientist to apply sophisticated hierarchically structured adaptive mesh layouts and functionality to existing structured grid codes that is minimally invasive. Our approach injects mesh and traversal management data layout and functionality into the application so that it resides between the layers that an application scientist is most likely to interact with (Figure 3.1). For the remainder of this section, although we consider these techniques to be general, we refer to PeanoClaw's design to provide a concrete example.

The foundation of the numerical scheme is a set of Clawpack and PyClaw kernels. Although a kernel k can be any function designed to run on a logically quadrilateral grid, k usually advances the solution in time by one time step according to the conservation laws governing the system.

$$k : (Q^{(n)}, Q^{(n,ghost)}, \Delta t) \mapsto Q^{(n+1)}. \tag{3.1}$$

The physical problem domain here is given by a (not necessarily smooth) mapping to a regular Cartesian computational grid Ω_h. This computational grid is supplemented by a ghost region $\Omega_h^{(ghost)}$, a band of cells surrounding the computational domain. Each cell of the computational domain and the ghost region boundary is assigned a number

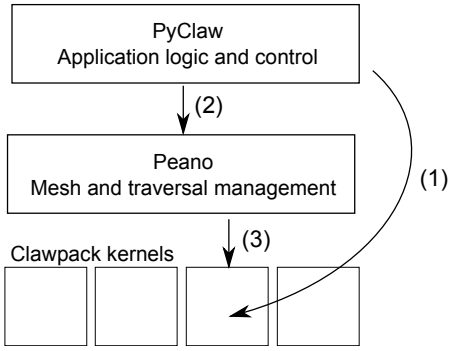


Fig. 3.1: Schematic overview of the three layer PeanoClaw architecture: the algorithm logic written in Python decides which kernels to run (1), it passes control to the grid management Peano (2) traversing the grid (multiple times), and Peano in turn calls back the selected kernels for each individual subgrid (3).

of unknowns; the solution values on the computational grid and the ghost region at time $t^{(n)}$ are denoted by $Q^{(n)}$ and $Q^{(n,ghost)}$, respectively. While the purpose of most kernels is to advance the solution by one time step, kernels for different purposes (initialization or visualization, e.g.) do exist.

The foundation of the mesh management and traversal is the C++ code Peano and its notion of a k-spacetree, a generalization of the well-known octree ($d = 2$) or quadtree ($d = 3$) concept [25, 27]. We embed the computational domain Ω into a square or cube, respectively. The geometric primitive then is cut into k pieces along each coordinate axis. We end up with k^d squares or cubes, respectively, that are all subsets of the original primitive, i.e. they establish a parent-child relation. For each primitive, we decide independently whether to continue with the splitting recursively.

Let the *level* of a primitive be the minimum number of refinement steps required to create it. The initial bounding box has level zero. All primitives of the same level are of the same size. As each primitive has exactly one parent, i.e. one predecessor, of k^d times the size, the overall data structure is a tree, a spacetree. We refer to the primitives as *nodes* of the spacetree $\mathcal{T}$, i.e. $\mathcal{T}$ is a set of nodes with a tree relation $\sqsubseteq_{child}$. $a \sqsubseteq_{child} b$ implies that a is a child of b, i.e. b is k^d times bigger than a and completely covers it. The tree's *root*, i.e. the bounding box, is $a_{root} \in \mathcal{T}$. An unrefined node, i.e. a node that has no children, is a *leaf*. Let $\mathcal{L} \subset \mathcal{T}$ be the set of leaves. If two nodes in $\mathcal{T}$ have the same level and share at least one vertex, they are *neighbors*.

Between the kernels and the application control logic, we inject the Peano mesh management and mesh traversal layer with its spacetrees. The key to minimizing invasiveness is embedding regular Cartesian PyClaw grids into the leaves of the adaptive spacetree.

$$\begin{aligned} f &: \mathbb{N}_0 \mapsto (\mathbb{N}, \mathbb{N}), \\ f(L) &= (cells, ghost)(L) \end{aligned} \tag{3.2}$$

prescribes a mapping of each leaf of the tree with level L onto a Cartesian equidistant grid Ω_h with $cells^d(L)$ cells and a ghost region around this grid of width $ghost(L)$ cells (Figure 3.2). Furthermore, we assign each spacetree node meta data $S(a)$ $\forall a \in \mathcal{T}$ holding grid properties used for local time stepping, e.g., on the subsequent pages.

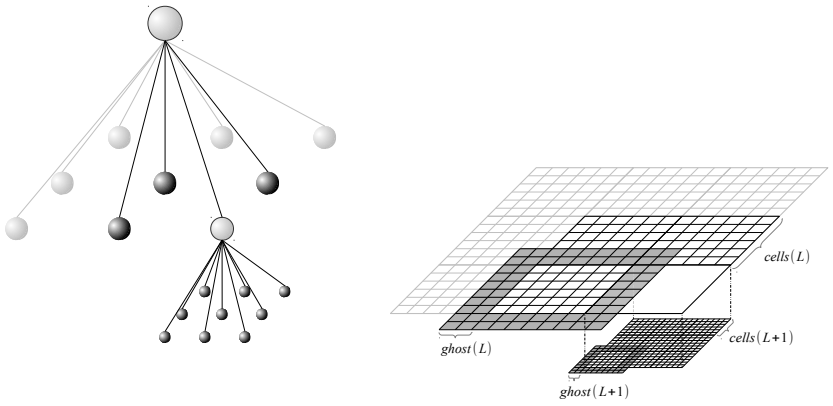


Fig. 3.2: Part of an adaptive mesh, including eight grids of level L and nine grids of level $L+1$. The sketch on the left illustrates the spacetree, while the right sketch visualizes the grids embedded into the spacetree nodes. The dark gray shaded areas depict the ghost layers. They are shown here for only two of the involved grids.

Each leaf $a \in \mathcal{L}$ holds the tuple $(Q^{(n)}(a), Q^{(n,ghost)}(a), Q^{(n+1)}(a), S(a))$, each refined node holds only meta properties, and all nodes of the same level have the same resolution (cell size)

$$\frac{k^{-level(a)}}{cells(level(a))} \tag{3.3}$$

along each coordinate axis. Our algorithms use $k = 3$ due to Peano but work for any $k \geq 2$. Obviously, a fully balanced k-spacetree with $\sum_l k^{dl}$ nodes yields a regular Cartesian PyClaw grid that is decomposed into subgrids each with ghost cells of its own, and every spacetree yields a set of non-overlapping subgrids of different resolution. These subgrids are coupled with each other due to their ghost regions that do overlap with neighboring grids, while the embedded grids can be passed directly to the low-level kernels. Peano is responsible for maintaining the set of grids as well as the adjacency information between these grids.

The application scientist will most frequently interact with the top layer of the decomposition, the PyClaw control layer. This Python scripting environment is responsible for managing the simulation parameters, logging, data input, output, and so forth. In a preamble, it prescribes the domain's size, its offset, as well as an initial grid, and it initializes all values. The algorithm itself consists primarily of a time stepping loop choosing kernels and triggering the mesh management to apply these kernels on all grids until all grids have advanced in time by a given Δt, i.e. whenever PyClaw has decided which kernel to use, it passes the control over to the Peano mesh layer (Figure 3.1). Peano sweeps the whole set of subgrids. For each subgrid, it invokes the kernel specified. Furthermore, data consistency operations and time step modifications are employed.

Mesh management and the solver layer are coupled via callbacks. Such a decomposition of responsibilities hides the mesh traversal order and logic in addition to PyClaw's decomposition into kernels from the user, while the kernel choice and the

algorithm control remain clearly exposed. From the perspective of the application scientist, this approach can be viewed as functional programming pattern where a user specifies which operations to perform on records while their arrangement, topology, and organization are hidden. Callback patterns resemble the design proposed for Array Building Blocks, e.g., and they are well-established for tree-based and, in general, mesh-based PDE solvers [8, 14, 21]. In PeanoClaw, the choice of execution order and the invocation of the operations is left to the Peano layer. Furthermore, the initialization of ghost layers due to (adaptive) neighboring subgrids is hidden from the kernels. The spacetree acts as compute data structure itself, and the mesh is embedded into this structure rather than extracted from the structure (cmp. [2]).

We reiterate that this approach allows us, with minimal modifications to the code, to extend an existing numerical software package with advanced adaptive data layout and functionality.

4. Time Stepping. Explicit discretizations of hyperbolic PDEs are typically stable under a time step size restriction of the form $\nu \leq \sigma$ where ν is the CFL number [15, 19]

$$\nu = s\Delta t/\Delta x \tag{4.1}$$

and σ is a factor depending on the particular discretization chosen. Here s is the maximum wave speed and Δx is the spatial cell width. For nonlinear problems, the value of s can change from one step to the next, and the CFL condition can only be checked after a step is taken. In the situation where the time step violates the CFL condition, this time step is rejected, and the kernel chooses a smaller time step satisfying the CFL condition. For problems involving multiple subgrids, PyClaw ensures per patch that the CFL constraint is met for each patch.

On an AMR grid, Δx is much smaller in fine regions than in coarse regions. In order to avoid the use of excessively small time steps in coarse regions, many AMR hyperbolic solvers advance coarse meshes to a given time using large steps while advancing finer meshes to the same time using smaller time steps. The ghost region values of the fine meshes are interpolated in both time and space from the coarser grids [5, 6, 7, 19]. This interpolation is complicated by the need to maintain conservation.

In PeanoClaw, we follow a similar approach, but we introduce two modifications, which are important if and only if conservation is required. First, we do not require coarse grids to cover regions where fine grids exist completely. Our approach thus exhibits a smaller memory footprint than totally overlapping approaches. Second, we preserve the course grid data consistency and conservation co-determined by finer grids on-the-fly.

Our spacetree methodology decomposes regions of the same refinement level into multiple grids. Hence, a global time stepping on one grid level is hard to realize and requires an all-to-all communication—if one node's grid decides to restrict the time step locally, all other grids of the same spacetree level that have already stepped would have to be rolled back and recomputed with the new time step. To avoid this, we decouple the time step restrictions also on each level, i.e. allow different grids on the same level to proceed with different time steps sizes. Conservation and the selection of grids to advance in time then have to be chosen carefully, but the gain in flexibility can pay off in runtime, as each grid can proceed with the maximum time step size allowed; this advantage has been noted previously in the context of unstructured grids [12].

Each grid holding $Q^{(n)}$ and $Q^{(n+1)}$ represents a time interval $[t^{(n)}, t^{(n+1)}]$ at any time (see [11] for a similar formalism). Algorithms of Berger-Colella-type requiring that coarser grids overlap all finer grids completely then obey the following invariants:

1. At the beginning, the initial condition $(t = t_0, \Delta t = 0)$ holds for any grid.
2. To advance the simulation by a given global Δt_0, the grids have to advance in time until $(t + \Delta t)(n) = t_0 + \Delta t_0$ for all grids n.
3. No grid ever is allowed to overtake its coarser grid, i.e. for $n' \sqsubseteq_{covered\ by} n$: $t(n') + \Delta t(n') \leq t(n) + \Delta t(n)$.
4. No coarse grid is allowed to proceed in time if a finer grid covered by this coarse grid has not caught up yet.

We replace the last two rules with versions similar to those of [12]:

3. No grid is allowed to advance in time if a neighboring grid has not caught up yet.
4. No grid is allowed to overtake its neighbor if the neighbor already is ahead.

Such a generalization does not distinguish whether adjacent subgrids have the same grid resolution or not, and it does not require the use of overlapping grids. At the same time, the constraints can be evaluated a priori and decide whether to advance a subgrid and which maximum time step size to allow. These decisions are hidden from the user and the kernels—the approach is minimally invasive.

5. Memory and Performance Model. Few simple models yield rough upper bounds for the memory footprint and the runtime behavior. For these models, we rely on a $\log(N/M) + \mu$ formalism used for example in [24], i.e. we assume that the spacetree construction refines regularly up to a certain level. In accordance with [24], N denotes the total of grid cells, M the number of cells per spacetree node, i.e. stems from $cells(L)$ in (3.2). A regular refined Peano spacetree has height $\log_{3^d} N/M$. Some regions then are refined further and add μ levels. The spacetree level $\log_{3^d} N/M$ is the coarsest level holding unknowns. All coarser levels in the spacetree do not hold unknowns.

We assume that, on any level greater or equal to $\log_{3^d} N/M$, a fraction of $f \in (0, 1)$ spacetree nodes is refined further. The assumption of a uniform f on all levels can be weakened by making f a vector. This is a technical modification yielding more accurate bounds, as, otherwise, f has to equal the biggest refinement factor throughout all adaptive mesh levels. Let m_0 and t_0 be the memory or time, respectively, that is needed to handle a given problem on a regular grid with the prescribed minimal mesh size, i.e. the grid corresponding to a spacetree of height $\log_{3^d}(N/M) + \mu$ where each node of level $L < \log_{3^d}(N/M) + \mu$ is refined. m_0 and t_0 are the benchmark values that have to be improved by PeanoClaw.

5.1. Memory Footprint. The memory footprint of the Berger-Colella scheme then is given by

$$\begin{aligned} m_{BC} &= 3^{-\mu d} m_0 + C f 3^{-(\mu-1)d} m_0 + C f^2 3^{-(\mu-2)d} m_0 + \dots \\ &\quad + C f^{\mu-1} 3^{-d} m_0 + C f^{\mu} m_0 \\ &< C m_0 \sum_{i=0}^{\mu} f^i 3^{-(\mu-i)d}, \end{aligned} \tag{5.1}$$

with the constant 3 stemming from the Peano spacetree scheme [25, 27]. The first summand $3^{-\mu d} m_0$ is the memory footprint induced by the coarsest level. The constant $C > 1$ represents the overhead introduced by ghost cells. Analogously, the memory

footprint of the present approach is given by

$$\begin{aligned} m_{PC} &= C(1-f)3^{-\mu d}m_0 + Cf(1-f)3^{-(\mu-1)d}m_0 + \dots \\ &\quad + Cf^{\mu-1}(1-f)3^{-d}m_0 + Cf^{\mu}m_0 \\ &= (1-f)Cm_0\left(\sum_{i=0}^{\mu} f^i 3^{-(\mu-i)d} + f^{\mu+1}\right). \end{aligned} \tag{5.2}$$

The $C > 1$ here is one plus the ratio of ghost grid cells to cells embedded into one spacetree node. As each subgrid is surrounded by a ghost layer, the constant here is greater than the constant in (5.1). PeanoClaw's memory footprint scales with $(1-f)$ relative the original Berger-Colella scheme due to (5.2).

5.2. Upper Bound for Runtime on Adaptive Grids. For the runtime estimates, we assume that the time step size on the spacetree nodes scales linearly with the mesh size, i.e. grids with a finer mesh size by a factor of three require three times more time steps. This is a pessimistic assumption, as the time stepping algorithm might proceed grids with smooth solution changes more aggressively.

The runtime of the Berger-Colella scheme then is bounded by

$$\begin{aligned} t_{BC} &= 3^{-\mu d}\cdot 3^{-\mu}t_0 + f3^{-(\mu-1)d}\cdot 3^{-(\mu-1)}t_0 + \dots \\ &\quad + f^{\mu-1}\cdot 3^{-d}\cdot 3^{-1}t_0 + f^{\mu}t_0 + \dots \\ &= t_0 \sum_{i=0}^{\mu} 3^{-(\mu-i)(d+1)} f^i + t_{conservation}\sum_{i=0}^{\mu} 3^i. \end{aligned} \tag{5.3}$$

While f again is the fraction of the refined spacetree nodes, the first factor in each term scales with the number of cells on the levels $log_{3^d} N/M$ through $log_{3^d} N/M + \mu$. They define the workload per time step. The second factor in each term introduces the linear dependency of the time step size on the mesh width of the respective grid. An additional term scaled with $t_{conservation}$ introduces the postprocessing steps starting from the finest grid that recover the conservation on the grid resolution boundaries. We assume $t_{conservation} \ll t_0$ and hence neglect it as we know that the postprocessing step acts on the adaptive boundaries—a submanifold.

The runtime of the present approach is bounded by

$$\begin{aligned} t_{PC} &\leq C(1-f)3^{-\mu d}\cdot 3^{-\mu}t_0 + Cf(1-f)3^{-(\mu-1)d}\cdot 3^{-(\mu-1)}t_0 + \dots \\ &\quad + Cf^{\mu-1}(1-f)\cdot 3^{-d}\cdot 3^{-1}t_0 + f^{\mu}t_0 + \dots \\ &= t_0 C(1-f)\sum_{i=0}^{\mu} 3^{-(\mu-i)(d+1)} f^i + t_{conservation}\sum_{i=0}^{\mu} 3^i. \end{aligned} \tag{5.4}$$

The constant $C > 1$ represents the overhead invested on the exchange of boundary data between adjacent grids of the same level and ghost layers and data from finer levels. In return, the speedup stemming from different patches of the same level proceeding with different time steps is neglected here.

The estimates in (5.3) and (5.4) suggest that we can expect at least a speedup scaling with $(1-f)^{-1}$ of the original Berger-Colella due to the present time stepping modifications if the implementation comes along with a reasonable small overhead. The runtime improvements due to AMR of both in (5.3) and (5.4) are by one dimension better than the memory improvement which results from the time step sizes co-determined by the grid sizes.

5.3. Local time stepping. We finally consider the performance effects of local time stepping, first in the context of a uniform grid and then for AMR. In the solution of nonlinear hyperbolic PDEs, the time step size Δt is usually chosen based on numerical stability considerations, as these are typically more restrictive than accuracy considerations. As mentioned in Section 4, the stability restriction takes the form of an upper bound on the CFL number $\nu = s\Delta t/\Delta x$, where s is the speed of the fastest waves occurring in the problem. Relation (3.3) determines Δx.

Typical PyClaw codes employ the same step size over the entire grid. Let s_m denote the maximal wave speed over the problem domain; then the entire grid is advanced at each step by $\nu\Delta x/s_m$ (note that s_m may vary from one time step to the next). This approach is potentially wasteful if there are large spatial variations in the maximal wave speed. For simplicity, suppose that the problem domain can be divided into two regions, with s_0 denoting the maximal wave speed over the first region and $s_1 < s_0$ denoting the maximal wave speed over the second region. Let f denote the fraction of the domain corresponding to the first region. Then an adaptive approach in which the second region is advanced using steps of size $\nu\Delta x/s_1$ yields a reduction in computational time by a factor of

$$f + (1 - f)\frac{s_1}{s_0}.$$

This factor can be significant if wave speeds are relatively small over most of the domain but much larger over a small region. This is also a type of situation in which AMR may be useful, and the advantage of adaptive time stepping may be even more pronounced in combination with AMR as we explore below. Let us adopt a less simplified model in which s_j denotes the wave speed in subgrid j and this value is again assumed constant in time. Then the speedup achieved by using spatially adaptive time stepping is

$$\bar{s} = \frac{\sum_{j=1}^{G} s_j}{G s_m}$$

which is just the mean wave speed divided by the maximum wave speed. G is the number of subgrids.

Since the stable time step size is proportional to (3.3), it is typical to refine by the same factor in both time and space on AMR grids. This strategy works well when the wave speeds are nearly uniform over the grid. However, if the wave speeds vary substantially in space, then spatially adaptive time stepping is beneficial. Using the AMR model considered in the previous section, and supposing that the mean wave speed over the ith level of refinement is $\bar{s}_1$, the runtime is bounded by

$$t_0 C(1 - f) \sum_{i=0}^{\mu} 3^{-(\mu-i)(d+1)} f^i \frac{\bar{s}_1}{s_m}.$$

Consider, for instance, the case of tsunami simulations using the shallow water equations. The wave velocities in the shallow water equations are

$$s = u \pm \sqrt{gh},$$

where u is the fluid velocity, g is the gravitational constant, and h is the fluid depth. In most cases, $u \ll \sqrt{gh}$ so $|s| \approx \sqrt{gh}$. In the deep ocean, typical depths are on the

order of thousands of meters, while near the shore they are only meters or tens of meters. Thus the wave speeds near the shore (where most of the refined regions lie) are typically ten times (or more) smaller than those in the deep ocean [19]. Spatially adaptive time stepping yields a dramatic speedup in this case, even though some overhead constants might enter the estimates from above in reality.

Different to the present approach, AMRClaw, e.g., enables the user to choose a temporal refinement factor different from the spatial refinement factor, in order to partially avoid the inefficiency described above. While this freedom on the one hand might reduce the time-to-solution, it bears, on the other hand, the risk to make the system unstable or to introduce unphysical dispersion. PeanoClaw's time stepping choice is robust.

6. Implementation. The present approach connects PyClaw and Peano due to (3.2) and relies on a $(k = 3)$-spacetree holding all data. While the resolution of the individual grids as well as the size of the ghost region can differ from spacetree level to spacetree level, let the constraints

$$\forall L \in \mathbb{N}: \quad cells(L) \geq 1, \tag{6.1}$$

$$\forall L \in \mathbb{N}: \quad 1 \leq ghost(L) \leq cells(L), \quad \text{and} \tag{6.2}$$

$$\forall L \in \mathbb{N}: \quad 3\ cells(L+1) > cells(L) \tag{6.3}$$

hold. We enforce that the grids embedded into the spacetree node are not empty (6.1) and that the ghost regions are not empty and they do not overlap with more than one neighbor along each coordinate axis (6.2). Each additional refinement level of the spacetree yields a finer grid (6.3). In practice, multiples of six and an L-independent choice of f are convenient, as this ensures that coarse grid cell faces and vertices coincide with fine grid cell faces and vertices. Furthermore, coarse grid cell centers coincide with fine grid cell centers—an advantage of three-partitioning compared to classical bipartitioning [16].

6.1. Mesh Processing Properties. With PyClaw specifying operations on the data in terms of a kernel operator, we propose to run the kernel on the individual grids embedded into the spacetree sequentially. While such a composed operator is specified multiplicatively, we enforce an additive constraint. For two nodes $a, b \in \mathcal{L}$ with a, b representing the associated solutions as well as the ghost data, the evaluation order shall be irrelevant, i.e.

$$k(a, \Delta t) \cdot k(b, \Delta t) = k(b, \Delta t) \cdot k(a, \Delta t). \tag{6.4}$$

Formally, grids can be read as independent tasks—one approach to parallelize the resulting scheme. In particular, kernel operations are commutative and cannot influence each other. This specification mirrors PyClaw's data flow paradigm: Here, the original grid together with the ghost region are mapped onto new grid values due to k in one step similar to a Jacobi. If the grid is split into two parts logically which mirrors loop reordering on a higher abstraction level, this split does not alter the mapping's image—a constraint enforced by (6.4).

6.2. Grid Traversal. With the grid specification and the sketch of the traversal's idea at hand, it is possible to formalize the traversal algorithm. Let the control layer prescribe a time step size Δt, a kernel k, and the three mappings $map, \hat{k}$, and $boundary$. Only the combination of map and $\hat{k}$ is an extension to existing PyClaw

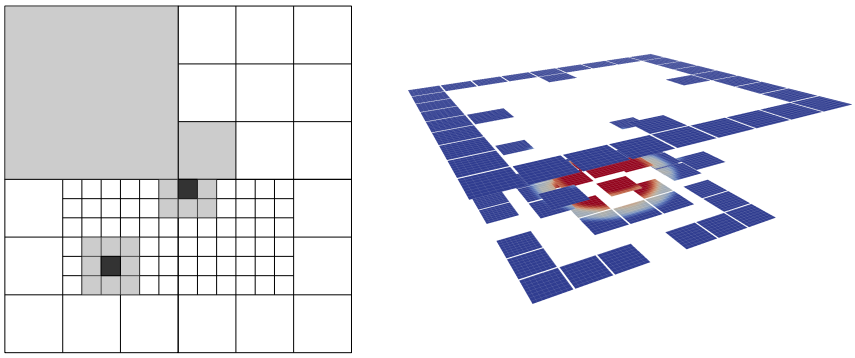


Fig. 6.1: On the left two different neighbor relations are shown. The lower left marked cell has eight neighbors which all reside on the same level. The upper right marked cell has only seven neighbors on three different levels.
A regular Cartesian grid is represented by a regular spacetree, i.e. each of the 9×9 subgrids in the image refer to one spacetree node holding a 16×16 grid itself. The vertical translation of the grids represents their current time stamp, i.e. the grids in the corners already have advanced further than the interior grids.

solvers that is logically broken up into two distinct operations. *map* acts as a mapping from one grid to another grid, while $\hat{k}$ comprises a veto mechanism selecting which grids to update and what the maximum time step is. For this, it acts on the grid meta data. For both operations, we provide default implementations realizing a conservative time stepping. The *boundary* mapping provides the functionality to fill the ghost layer of the global grid. All arguments are passed to Peano. Peano runs through the spacetree depth-first using the helper operation $neighbor : \mathcal{T} \mapsto \mathcal{P}(\mathcal{T})$ with the image power set $|neighbor| \leq 2^d$ or $|neighbor| \leq 3^d - 1$. Formally, *neighbor* accepts a spacetree node and returns up to $3^d - 1$ spacetree nodes that have a maximal level smaller or equal to the preimage's level and are adjacent to the input argument. Alternatively, it studies only neighboring nodes of the same or coarser size that share a common face. If one of these nodes is a refined node, the corresponding *neighbor* entry holds only a state as no grid is available for refined nodes. If a neighbor does not exist on the same level, the mapping searches for a neighbor on the next coarser level recursively (Figure 6.1). For the root node, no neighbors exist. Kernels including transverse wave propagation require a neighborhood of up to $3^d - 1$ nodes, other kernels get along with neighbors only connecting due to a common face. The latter realize a pattern called dimensional splitting.

PeanoClaw's grid traversal is given by Algorithm 1. Its rationale are straightforward: The traversal sweeps over the whole spacetree and passes the individual grids embedded into the nodes to the kernel. As the traversal is top-down depth-first, the traversal immediately descends if a refined spacetree node is encountered. Afterwards, it updates the meta data which is analyzed bottom-up due to $\hat{k}(\ldots, \top)$. $\hat{k}$'s last parameter indicates whether the traversal steps down with $\top$ and $\bot$ denoting yes and no. As the spacetree is traversed depth-first, the grid levels are updated from coarse

Algorithm 1 Peano traverses the spacetree top-down, and invokes PyClaw routines as call-backs throughout this traversal.

procedure TRAVERSE($\Delta t, k, boundary, \hat{k}, map$)
  $\Delta t(S(a_{root})) \leftarrow \Delta t$
  TRAVERSE($k, boundary, \hat{k}, map, a_{root}, \bot$)
procedure TRAVERSE($k, boundary, \hat{k}, map, a, a_{parent}$)
  $x \leftarrow \top$ ▷ helper variable
  if $a_{parent} \neq \bot$ **then**
    $(S(a), x) \leftarrow \hat{k}(S(a), S(neighbour(a)), S(a_{parent}), \top)$
  if $a \in \mathcal{L}$ **then**
    if $x = \top$ **then**
      $Q^{(n)}(a) \leftarrow Q^{(n+1)}(a) \cup Q^{(a,ghost)}(n)$ ▷ Switch to next time step
      $Q^{(n)}(a) \leftarrow boundary(Q^{(n)}(a), t(S(a)))$ ▷ Set boundary conditions
      for all $a' \in neighbour(a)$ **do**
        $Q^{(n,ghost)}(a) \leftarrow map(Q^{(n,ghost)}(a),$
          $Q^{(n)}(a'), Q^{(n+1)}(a'), t(S(a)), t(S(a')))$
          ▷ Update ghostlayer from neighbours
      $k(a, \Delta t(S(a)))$ ▷ Invoke PyClaw kernel
        ▷ May modify $\Delta t(S(a))$
      for all $a' \in neighbour(a)$ **do**
        $Q^{(n+1)}(a') \leftarrow map(Q^{(n+1)}(a'),$
          $Q^{(n)}(a), Q^{(n+1)}(a), t(S(a')), t(S(a)))$
          ▷ Modify fluxes on coarse grid
        $Q^{(n,ghost)}(a') \leftarrow map(Q^{(n,ghost)}(a'),$
          $Q^{(n)}(a), Q^{(n+1)}(a), t(S(a')), t(S(a)))$
          ▷ Update neighbours
  else
    for all $a' \sqsubseteq_{child} a$ **do**
      TRAVERSE($\Delta t, k, boundary, \hat{k}, map, a', a$)
      $S(a) \leftarrow \hat{k}(S(a), S(a'), \bot)$

to fine. As the spacetree is traversed depth-first, the mapping *neighbor* for each node can be built-up recursively and on-the-fly. It is not persistent. Its technical realisation is not presented in the algorithm. As the spacetree is traversed depth-first, nodes may inherit meta data such as the time step size due to a mapping $\hat{k}(\ldots, \bot)$. For all grid levels, all grid and ghost layer sizes are given by (3.2).

Algorithm 2 A wrapper around the spacetree traversal implements the task-like time stepping, i.e. it loops until all patches have reached the terminal time.

procedure TRAVERSE($\Delta t, \ldots$) ▷ Wrapper around operation from Algorithm 1
  Memorize global $\Delta t_0 = \Delta t$
  $\forall a \in \mathcal{T} : t(S(a)) \leftarrow 0$
  $\forall a \in \mathcal{T} : \Delta t(S(a)) \leftarrow 0$
  while $\exists a \in \mathcal{T} : t(a) + \Delta t(a) < \Delta t_0$ **do**
    TRAVERSE($\Delta t(a_{root}), \ldots$) ▷ Call original operation from Algorithm 1

6.3. Time Stepping. PeanoClaw supports local time stepping and hides most of the time step management: We assume that each kernel call advances the grid by a time step smaller than or equal to the passed time step size. As soon as the kernel terminates, the actually chosen time step size is stored in the argument Δt, i.e. Δt is an in and out argument. Each node of the spacetree is assigned the tuple $(t, \Delta t)$ stored within the grid's meta data. We often write $t(a)$ instead of $t((S(a))$ and $\Delta t(a)$ instead of $\Delta t(S(a))$. Furthermore, the state also encodes the grid's level and whether the spacetree node is a leaf as well as the bounding box corresponding to the spacetree node. The mappings $\hat{k}$ and *map* then implement the conservative Berger-Colella scheme—*map* either sets ghost values on the interfaces of finer and coarser grids, or *map* coarsens a fine solution onto the ghost cells of adjacent coarser grids, or *map* updates a coarser grid's solution due to flux between interfaces of fine and coarse grids—whereas the kernels k and *boundary* remain unchanged. Finally, the whole traversal routine (Algorithm 1) is wrapped by an outer loop (Algorithm 2).

6.4. *map***.** The operation *map* has three different objectives: First, it couples grids of the same level. Second, it projects data from coarse levels to finer levels where they act as boundary conditions. Third, it projects fine grid updates to the coarser levels. The latter mapping preserves the conservation laws and it paves the way for the next time steps on the coarser levels or on neighboring cells. *map* is always passed data from two different grids: the image node a, i.e. the node whose data can be manipulated, and the preimage node b (cf. Algorithm 1).

$$Q(a) \leftarrow map(Q(a), Q^{(n)}(b), Q^{(n+1)}(b), S(a), S(b)),$$

so $Q(a)$ is both input and an output data. It can be either the old or new data structure or a part of the ghost cells. $S(a)$ and $S(b)$ are meta data belonging either to a or b. The implementation of *map* for the time stepping reads as follows:

- If $b \notin \mathcal{L}$, return immediately without modifying $Q(a)$.
- If node a and the preimage node b are on the same level and $t(a) + \Delta t(a) = t(b) + \Delta t(b)$, b is a neighbor and the neighbor has not advanced in time yet but will do in this iteration. Copy its new values, i.e. values from $Q^{(n+1)}(b)$, into the own region $Q(a)$. The previous values of $Q(a)$ are overwritten. In this case, $Q(a)$ equals a ghost layer due to Algorithm 1.
- If a node a and the preimage nodes b are on the same level and $t(a)+\Delta t(a) < t(b) + \Delta t(b)$, b is a neighbor and the neighbor has already advanced in time. Interpolate the neighbor's values $Q^{(n)}(b)$ and $Q^{(n+1)}(b)$ linearly in time onto $Q(a)$. In this case, $Q(a)$ equals a ghost layer due to Algorithm 1. Different to the original algorithm, we do not require two adjacent patches to process in time simultaneously. Hence, this inequality check is important. The other way round, the inequality may never occur. If a grid is already ahead of its neighbors, the veto mechanism of $\hat{k}$ should forbid this grid to advance in time once more.
- If the preimage node b of a node a is assigned to a coarser level, b is a neighbor and we have to interpolate the neighbor's data $Q^{(n+1)}(b)$ and $Q^{(n)}(b)$ both in space and time onto the local grid's ghost layer. The local values $Q(a)$ are replaced by the interpolated values.
- If the preimage node b of a node a is assigned to a finer level than a, we have to restrict the fine grid data $Q^{(n+1)}(b)$ and $Q^{(n)}(b)$ onto the image grid.
  - If the image grid and the preimage grid do overlap, i.e. if the preimage is to be restricted onto a ghost layer ($Q(a)$ is a ghost layer), add the space-

averaged value of $Q^{(n+1)}(b) - Q^{(n)}(b)$ to the cells of $Q^{(n,ghost)}(a)$: for each cell of $Q(a)$, we identify which cells of $Q^{(n)}(b)$ or $Q^{(n)}(b)$ respectively overlap, and compute the (weighted) average. The plus before $Q^{(n+1)}(b)$ ensures that the newest values are restricted to the destination grid. The minus before $Q^{(n)}(b)$ ensures that the contributions from the older time step that has been restricted to the destination grid before are removed.
- If the image grid and the preimage grid do not overlap, i.e. if the preimage is to be restricted onto the image's new values, add the fluxes to $Q(a)$ to preserve the conservation along the grid boundaries. For this, the scaling $\frac{\Delta t(b)}{\Delta t(a)}$ has to be taken into account.

The latter two steps are a modification of the classical time stepping algorithms, as effects of the fine grid are immediately restricted to the coarse patches instead of a global postprocessing step as suggested in [7]. In [3, 5, 6, 7, 10] and references therein it is discussed how the conservation along the grid interfaces is preserved.

6.5. $\hat{k}$**.** The operation $\hat{k}$ has three objectives: First, it vetoes the update of individual patches. Second, it updates the grid meta data—in particular for spacetree nodes that do not hold grids as they are refined. For this, it is important to know whether $\hat{k}$ is evaluated due to a spacetree automaton descend or ascend. Third, it precedes the time step.

$$\hat{k} : S \times S^{3^d-1} \times S \times \{\top, \bot\} \mapsto S \times \{\top, \bot\}$$
$$(S(a), x) \leftarrow \hat{k}(S(a), S(n), S(pc), td)$$

while $S(a)$ is both an input and an output data structure. $S(n)$ is the set of meta properties of the neighbors, td indicates whether $\hat{k}$ is called throughout a traversal step down or a step up, and $S(pc)$ is the meta data of the parent if $td = \top$ or the meta data of a child if $td = \bot$. The implementation of $\hat{k}$ reads as follows:

- If the descend flag holds ($td = \top$) and if the preimage is unknown, i.e. it belongs to the parent $\bot$, a is the spacetree's root, i.e. $a = n_{root}$. The result flag is $x = \bot$, $S(a)$ remains unchanged.
- If the descend flag holds ($td = \top$), the boolean results flag x is true ($\top$) if and only if

$$t(a) + \Delta t(a) \leq t(n') + \Delta t(n') \ \forall n' \in S(n) \qquad \text{and}$$
$$t(a) + \Delta t(a) < t(pc) + \Delta t(pc).$$

- If $x = \top$, and if $a \in \mathcal{L}$, update the current grid's time stamp to $t(a) \leftarrow t(a) + \Delta t(a)$ and set, afterwards, $\Delta t(a) = \min_{n' \in S(n)}(t(pc) + \Delta t(pc) - t(a), t(n') + \Delta t(n') - t(a))$ (cf. [12]).
- If $x = \top$, and if $a \notin \mathcal{L}$, update the current grid's time stamp to $t(a) \leftarrow t(a) + \Delta t(a)$.
- If the descend flag holds, the state's time step size is finally always set to $\Delta t(a) = t(pc) + \Delta t(pc) - t(a)$. The result value x is not modified.
- If the descend flag does not hold ($td = \bot$), we update the state's time step size according to $\Delta t(a) \leftarrow min(\Delta t(a), t(pc) + \Delta t(pc) - t(a))$.

The last step updates the time step size of the coarse node depending on the time step taken by one of its child nodes. Since Algorithm 1 performs this step for all children, the resulting coarse node's time step is the minimum of all its children.

6.6. Grid Initialization and Control. The grid interaction paradigm enabling the user to manipulate the adaptive mesh picks up the minimal invasive philosophy. PeanoClaw's initial grid is set up due to $h > 0$. It is a regular Cartesian grid, i.e. a perfectly balanced spacetree of height ℓ with

$$\ell = \min_{L \in \mathbb{N}} \left(\frac{1}{3^L \cdot cells(L)} < h \right).$$

The inequality mirrors the mesh width computation (3.3).

For mesh manipulation and dynamic adaptivity, we add the PyClaw kernel calls (3.1) an optional out argument. The kernel's signature expects the mesh width of the grids as input argument. While PyClaw neglects modifications of this argument, PeanoClaw does evaluate it as soon as the kernel has terminated. If the kernel returns a mesh width smaller than the input value, PeanoClaw adds additional spacetree levels throughout the subsequent traversal. If the returned mesh width is coarser than the input value, PeanoClaw coarsens the grid.

If new subgrids are added to the spacetree, their cell values are initialized due to a d-linear interpolation. If a grid is removed, a new coarser subgrid replaces $(k^d)^i$ finer grids. $i > 1$ takes into account that multiple nodes of the spacetree might be removed in one step. The values of the new coarser leaf stem from averages of the finer grid, i.e. all values of cells covered by one new coarse grid cell are summed up and scaled. These two inter-grid operations are set by default. The user can modify them and inject tailored kernels instead of.

7. Experiments. All experiments reported here were conducted on a commodity Intel Core i7 machine equipped with eight GByte of RAM and running at 3.4 GHz.

We consider two applications, both in two spatial dimensions. The first is the shock-bubble interaction problem introduced already in Section 2. The second is a model of a circular breaking dam. It involves the shallow water equations:

$$\begin{aligned}
&d = 2,\ \Omega = (0,1)^d, h, u, v : \Omega \times \mathbb{R}_+ \mapsto \mathbb{R} \text{ with } (h,u,v)(x,t) \text{ given by} \\
&\qquad h_t + (hu)_x + (hv)_y = 0, \\
&(hu)_t + \left(u^2 + \frac{1}{2} g h^2 \right)_x + (huv)_y = 0, \\
&(hv)_t + (huv)_x + \left(v^2 + \frac{1}{2} g h^2 \right)_y = 0, \qquad \text{and} \\
&\qquad h(t=0,\vec{x}) = \begin{cases} 2 & \text{if } \sqrt{(x-0.5)^2 + (y-0.5)^2} \leq \frac{1}{4} \\ 1 & \text{otherwise} \end{cases}
\end{aligned} \tag{7.1}$$

on a unit square domain with reflecting boundaries for $t \geq 0$ and g being the acceleration of gravity.

For all scenarios, we use the second order explicit Clawpack discretisation [17]. Due to the computational stencil

$$\begin{bmatrix} \cdot & \cdot & * & \cdot & \cdot \\ \cdot & \cdot & * & \cdot & \cdot \\ * & * & * & * & * \\ \cdot & \cdot & * & \cdot & \cdot \\ \cdot & \cdot & * & \cdot & \cdot \end{bmatrix}$$

all experiments use ghost layer of size two. If dimensional splitting is turned on, $\cdot$ becomes zero. Otherwise, the stencil is full.

The operation $\hat{k}$ is a direct realization of the properties discussed in Section 6.5, while *map* distinguishes three cases. It is a linear or bilinear interpolation operator if it maps a coarse grid onto a finer grid. In this case, the fine grid is a ghost region. It is an averaging stencil

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

if it maps a fine grid to a coarser grid and the image overlaps the preimage. In this case, the coarser grid is the ghost region of a coarser subgrid adjacent to the preimage grid. It is finally a conservation-preserving update according to [19] if it maps a finer grid to a coarser grid and image and preimage do not intersect. In the latter case, *map* fixes the flux on a grid resolution interface, i.e. it recovers mass conservation. For each time step of a fine grid the values of the cells of the neighboring coarser grid u_{coarse} are modified as follows:

$$q_{coarse} \leftarrow q_{coarse} + \frac{1}{h_{coarse}}\left(\Delta t_{fine} \cdot \frac{F_{fine}}{r} - F_{coarse}\right).$$

Here F_{fine} is the flux from the fine cell to the coarse cell regarding to the current fine grid values. In contrast to this, F_{coarse} is the flux from the coarse cell to the fine cell, regarding the current coarse grid values.

In accordance with the ghost layer interpolation of *map*, all unknowns of any newly created node's grid hold the bilinear interpoland of the previous, coarser solution. Whenever nine grids are merged into one coarser grid, each coarse grid unknown holds the average of all fine grid unknowns covered by this coarse grid cell.

7.1. Local Time Stepping on Regular Grids. We first study the effect of local time stepping while using a uniformly refined, static grid. We examine the temporal advancement of different subgrids in the radial dam-break problem. We focus on the subgrids lying along the diagonal $x = \frac{k}{10} \cdot (1,1)^T \in \Omega$, as depicted in Figure 7.1). The 9×9 subgrids hold 4×4 cells each (Figure

Since the wave speeds ahead of the shock are smaller than those behind the shock, the local time stepping constraints (cf. Section 4) allow subgrids to advance rapidly in time until the shock wave reaches them. The grids in the corner ($k = 1$) first perform a large time step, and the grids near the breaking dam ($k \in \{4, 5\}$) then catch up. After 22 iterations—around half of the total number of cells in the grid along the diagonal, i.e. scaled by $\sqrt{2}$—the shock approaches the outer corner. Now, the time step size in the corner grids is chosen smaller, and the outer grids drag behind the grids near the center.

We next study both test problems with a full stencil on regular grids of size 162×162, 486×486, and 1458×1458. They are represented by regularly refined Peano spacetrees. For the grids embedded into the spacetree nodes, we study different settings starting from the whole grid embedded into one spacetree node to a 6×6 grid embedded into each spacetree node. If the whole grid is embedded into one spacetree node, the present algorithm resembles the original PyClaw time stepping.

CFL allows grids resolving regions of low activity to proceed in time faster than other grids. This flexibility saves total computations measured in cell updates (Figure 7.2). The savings due to the decoupling are the bigger when smaller subgrids are used.

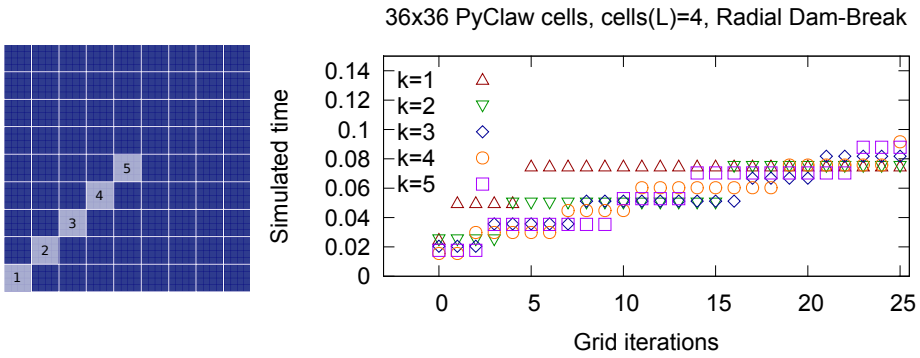


Fig. 7.1: Solution is tracked within different subgrids of the regular grid (left), and the subgrids overtake each other throughout the simulation when the breaking dam front rushes through the grid (right).

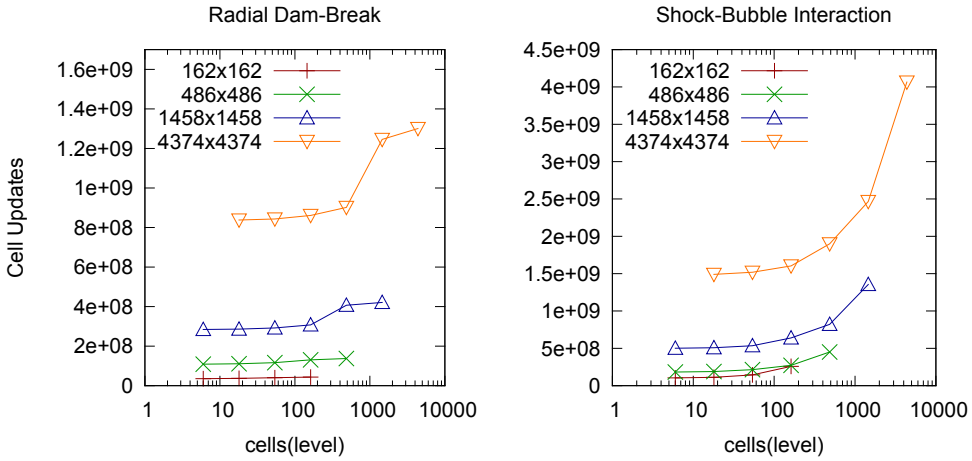


Fig. 7.2: Number of cell updates depending on the chosen subgrid size for different global grid resolutions.

Since each cell update corresponds to roughly the same amount of arithmetic work, we expect walltime to be proportional to the number of cell updates in Figure 7.3. However, the use of smaller subgrids also leads to an increase in overhead. This overhead is determined by multiple factors, including data movement for ghost cell updates over cache blocking effects and callbacks between different languages and components. As a consequence, there exists an optimal grid size that is small enough to take advantage of local time stepping in order to reduce the number of cell updates, but big enough to avoid excessive overhead.

The best case overall computational savings compared to pure PyClaw runs increase with the size of the full grid. For the two present experiments and the different grid settings, a subgrid size of around 128×128 yields a reasonable speedup. By

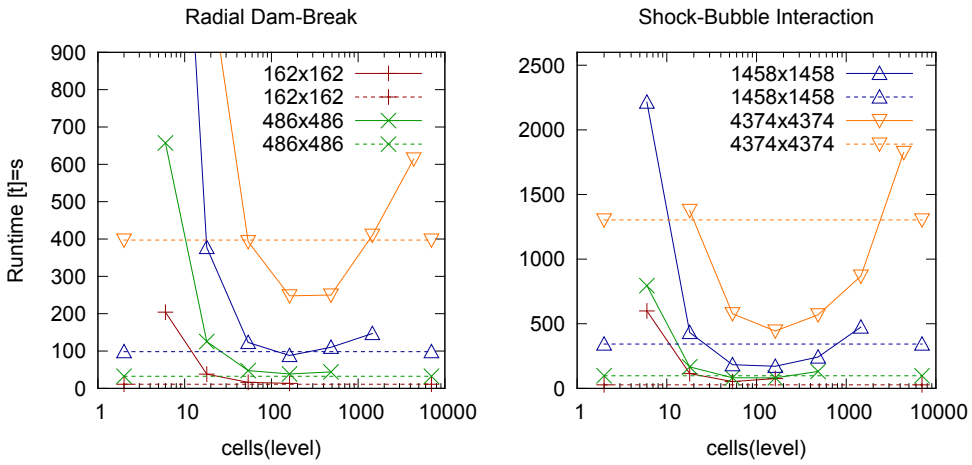


Fig. 7.3: Runtime for different subgrid sizes. Dotted lines are pure PyClaw runs, the other measurements result from PeanoClaw.

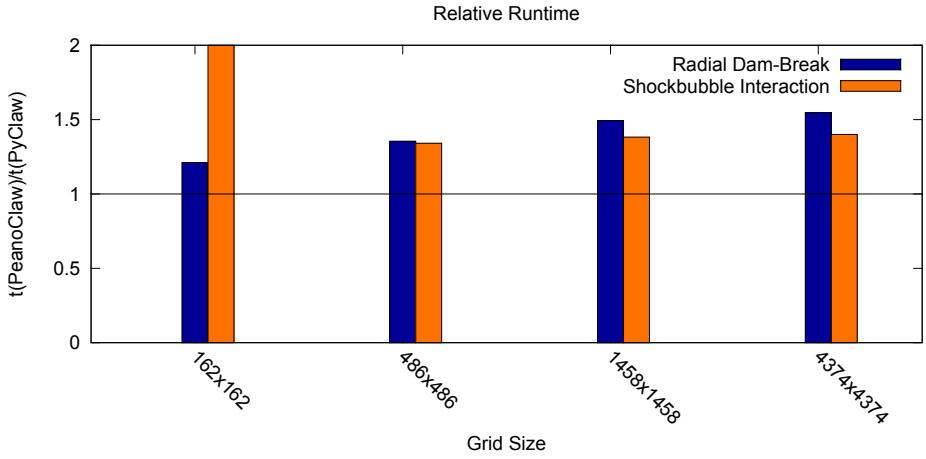


Fig. 7.4: Performance overhead for PeanoClaw holding only one node with one embedded grid. Measurements normalized by pure PyClaw runs.

properly selecting the subgrid size, we speed up the simulation by up to a factor of two for the radial dam-break and more than a factor of two for the Shock-Bubble interaction experiment. The latter results coincide with results reported for geophysical applications [10].

If we embed the whole computing grid into one spacetree node only, i.e. if the spacetree has height zero, the resulting code is not as fast as PyClaw. This is due to the aforementioned overhead, measured in Figure 7.4. For reasonably sized grids, PeanoClaw's per-subgrid performance is $60 - 80\%$ of the pure PyClaw kernels, and it depends on the computational workload per cell. The Euler flow is computationally

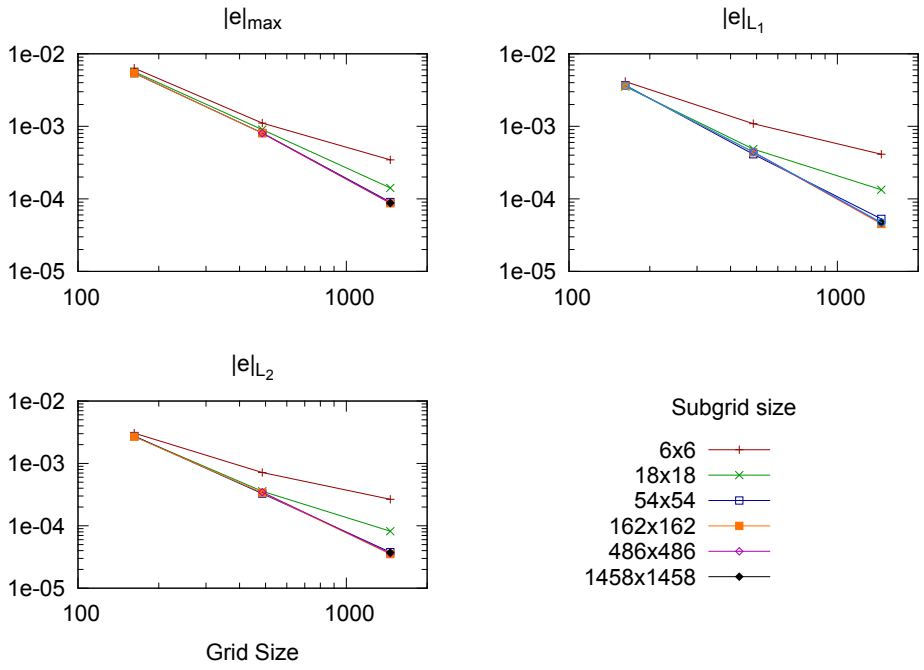


Fig. 7.5: Global error in the maximum (top left), L_1 (top right) and L_2 (bottom left) norm for the shallow water experiment with smooth initial conditions.

more expensive than the shallow water kernel and thus comes along with a smaller per-patch overhead. In the following we will concentrate on the (more challenging, in terms of PeanoClaw performance) shallow water simulation.

We also studied the speed of simulations using dimensional splitting; in most cases, the speed was similar. However, some cases ran significantly faster when dimensional splitting was used. Dimensional splitting in this context requires further studies. For the remaining results, we consider only the unsplit Clawpack algorithms.

7.2. Accuracy of Local Time Stepping on Regular Grids. Next we investigate the accuracy of PeanoClaw's local time stepping. We consider the the shallow water equations from (7.1) with a smooth initial condition:

$$\begin{aligned} u(x,y) &= 0 \\ v(x,y) &= 0 \\ h(x,y) &= \frac{1}{\sigma \cdot \sqrt{2\pi}} \cdot e^{-50r^2} \\ \text{with} \quad r^2 &= (x-0.5)^2 + (y-0.5)^2. \end{aligned}$$

As reference value, we ran the setup with a mesh size four times as fine as the finest studied problem, i.e. for a $(1458 \cdot 4) \times (1458 \cdot 4)$ grid, and all errors are given with respect to this run.

We observe that PeanoClaw's local time stepping does not harm the second order accuracy as long as the ratio of subgrid sizes to global grid is sufficiently big (Figure

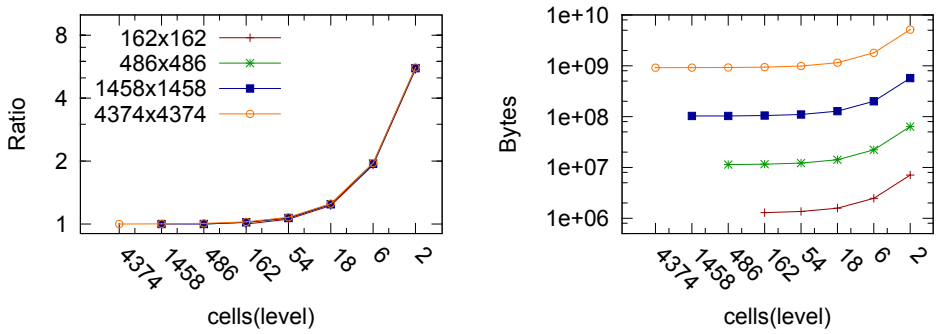


Fig. 7.6: Left: Ratio of PeanoClaw's memory footprint to PyClaw's memory footprint for four different global grid sizes with different subgrid choices ($cells(level)$). Right: This determines the memory footprint of a regular grid handled by PeanoClaw.

7.5). Subsequent experiments reveal that PeanoClaw preserves PyClaw's accuracy for non-smooth settings, too.

7.3. Memory Overhead on Regular Grids. A final study with regular grids examines the memory behavior for global grids with the resolution 162×162, 486×486, 1458×1458, and 4374×4374. According to Figure 7.6, the memory footprint per subgrid depends linearly on twice the number of cells within the subgrid (as we hold current and next solution) plus the cells of the ghost region that is a $d-1$ dimensional submanifold: with a given fine grid level ℓ, it increases approximately by a factor of

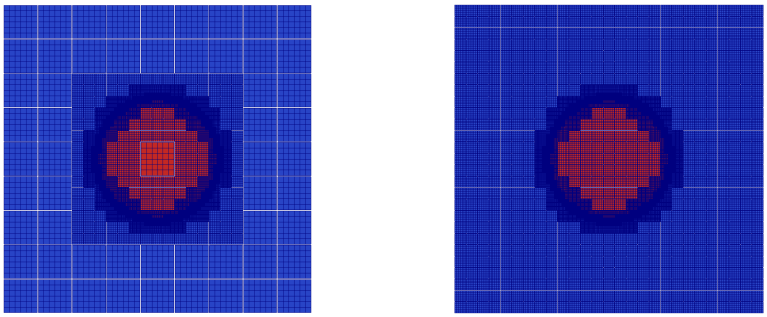
$$\frac{(cells(\ell)+2ghost(\ell))^d + cells^d(\ell)}{2cells^d(\ell)}.$$

The memory requirements for the meta data and the PyClaw control layer are insignificant. The relative overhead does not depend on the total grid size. Smaller subgrids impose a greater memory overhead, and when the subgrid size is of the same order as the discretization stencil, the overhead becomes very large. However, as can be seen in Figure 7.3 and Table 7.4, the best computational performance is achieved by using significantly larger subgrids, for which the additional memory overhead is quite small.

7.4. Speedup and Memory Savings on a Static Adaptive Grid. To analyse PeanoClaw's adaptive grid realisation, we study a static adaptivity pattern for the radial dam-break, $cells \equiv 6$, and the finest cell width $\Delta x = 1/4374$. Let the finest mesh cover the discontinuity. We stop the simulation before the shock front leaves this subgrid.

Around the very fine subgrid having a level of six we coarsen the spacetree up to a given minimal level as aggressively as the tree structure allows. Let μ denote the number of levels of coarsening allowed; i.e., the difference between the minimal and maximal level of refinement (cf. Section 5). In our example the coarsest level possible corresponds to a value of two, in which case $\mu = 6 - 2 = 4$ (see Figure 7.7a). Of course, $\mu = 0$ denotes a uniform fine grid with no coarsening.

Adaptivity reduces the number of cells compared to a regular grid, and hence it reduces the memory footprint and the computational workload (Table 7.1). As



(a) $\mu = 4$ (b) $\mu = 3$

Fig. 7.7: Two statically refined grids with different μ.

Table 7.1: Normalized number of cell updates and normalised runtime for different μ whereas the finest mesh size always is set to $1/4374$. $cells \equiv 6$.

μ	#Subgrids	#Cells	Updates	Runtime	Averaged f
0	$5.31 \cdot 10^5$	$1.91 \cdot 10^7$	$1.20 \cdot 10^8$	2953.85	-
1	$6.46 \cdot 10^4$	$2.33 \cdot 10^6$	$6.46 \cdot 10^6$	347.87	0.01
2	$1.40 \cdot 10^4$	$5.06 \cdot 10^5$	$2.37 \cdot 10^6$	94.26	0.18
3	$8.86 \cdot 10^3$	$3.19 \cdot 10^5$	$2.18 \cdot 10^6$	67.61	0.25
4	$8.40 \cdot 10^3$	$3.02 \cdot 10^5$	$2.17 \cdot 10^6$	64.53	0.33

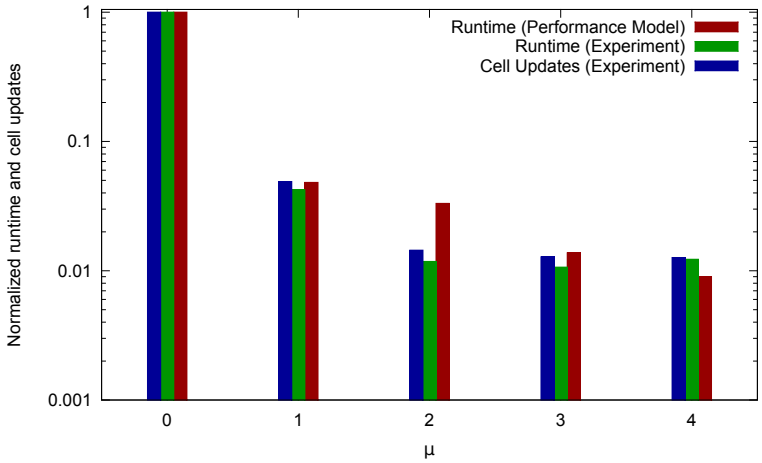


Fig. 7.8: Expected speedup due to the performance model compared to measured speedup and measured number of cell updates. All values are normalized with respect to the regular case.

adaptive grids allow coarse subgrids to advance in time faster than fine grids, the runtime reduction is more significant than the memory reduction. For both quantities, the coarsening of a fine grid has higher impact than the coarsening of an already rather coarse grid, while the cost per cell remains almost independent of μ, i.e. the total runtime scales with the cell updates.

This result is in accordance with the predictions in Section 5.2 (Figure 7.8). For the comparison, we choose the constants C in (5.2) and (5.4) such that all values normalized by regular grid runs are 1 for $\mu = 0$. They illustrate that the overhead to fill the ghost layers of adaptive meshes and to preserve the conservation along resolution interfaces is comparatively small though it gains impact for $\mu = 4$. For $0 < \mu < 4$, the adaptive time stepping renders the model prediction a pessimistic bound. While the AMR's memory advantages are predictable, the runtime impact is robust but depends on the concrete setting. With a validated performance model at hand, we reiterate that a choice of subgrid size six is in practice ill-suited as the AMR advantage cannot compensate the overhead studied in Section 7.1 and refer to realistic setups in the next section.

7.5. Dynamic Adaptivity. In this section we solve the radial dam-break problem using dynamically adaptive grids. This time, we stop the simulation when the shock hits the domain's boundary, and the finest grid follows the ring-shaped expansion of the two wave fronts: we know the wave speeds of both the rarefaction wave and the shock wave. They define a time-dependent ring-shaped domain around the domain's center. Whenever a subgrid intersects with this ring, it is refined up to the maximal allowed refinement level. Otherwise, we coarsen as aggressively as possible given the spacetree formalism.

In Figure 7.9, we present a histogram of cells and cell updates over the spacetree traversals that distinguishes total and per level measurements. Here, the finest level is the spacetree level four with $cells(L) = 6\ \forall L$. We identify two phases: First, the number of fine grid cells grows significantly. After approximately 1250 traversals, the fine cell and total cell number drop. We identify how the shock spreads before the rarefaction wave has propagated far enough to allow the center of the domain to coarsen again. It is clear that almost all the computational effort is spent in cell updates on the finest level of subgrids.

We restrict from now on to one fifth of the former simulation. Afterwards, the grid becomes rather regular and effects arising from highly inhomogeneous grid resolutions are not that clear anymore. We first track the error introduced by PeanoClaw relative to a regular PyClaw run with the finest grid resolution (Table 7.2) in the L_1 norm. Smaller subgrids give rise to larger errors. One possible reason for this is that neighboring grids may often take time step sizes that differ very slightly. Then one of the grids must take a very small step to catch up; the small step uses a very low CFL number, which leads to numerical diffusion. However, there may be other causes and this is an area for further study. As expected, in general the use of finer fine grids leads to greater accuracy. All choices of μ and subgrid size lead to reasonably accurate solutions.

Table 7.3 shows the number of cell updates for each simulation, relative to the number of updates required for a pure PyClaw run with a uniform grid of the finest mesh size. The first column of the Table ($\mu = 0$), corresponding to runs with no coarsening, shows that the local time stepping alone reduces the number of updates required by 30%–40%. Combining this with adaptive coarsening/refinement leads to reductions of up to a factor of ten in the number of updates that must be computed.

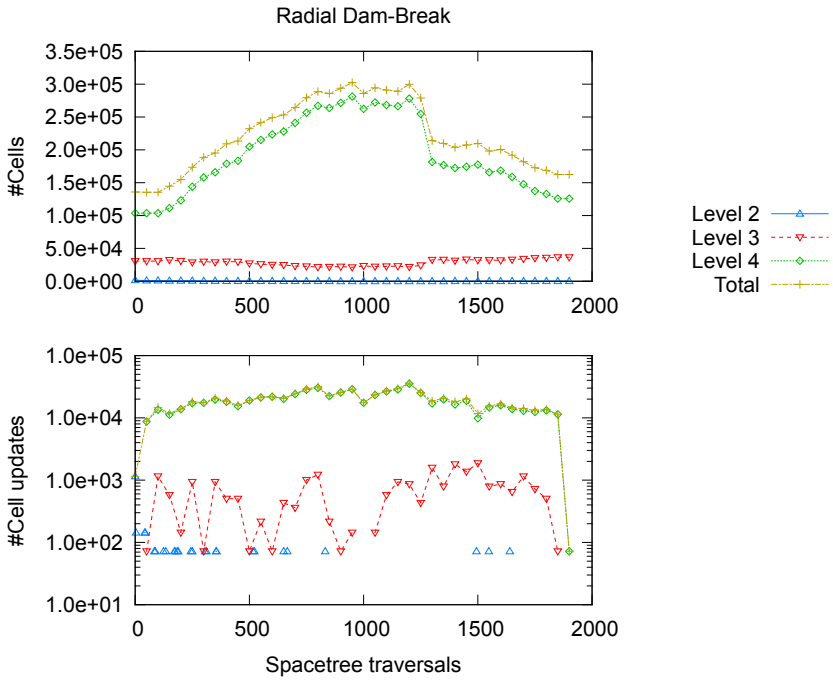


Fig. 7.9: Number of cells and cell updates over time.

Table 7.2: L_1 errors for different μ relative to regular grid simulations.

Finest Grid size	Subgrid size	Number of levels of coarsening			
		$\mu = 0$	$\mu = 1$	$\mu = 2$	$\mu = 3$
486×486	6×6	6.65×10^{-6}	6.65×10^{-6}	6.65×10^{-6}	
	18×18	3.47×10^{-6}	3.47×10^{-6}		
	54×54	5.74×10^{-6}			
1458×1458	6×6	5.11×10^{-6}	5.11×10^{-6}	5.11×10^{-6}	5.11×10^{-6}
	18×18	2.09×10^{-6}	2.09×10^{-6}	2.09×10^{-6}	
	54×54	9.63×10^{-7}	9.63×10^{-7}		
	162×162	9.23×10^{-7}			
4374×4374	18×18	9.64×10^{-7}	9.64×10^{-7}	9.64×10^{-7}	9.64×10^{-7}
	54×54	3.98×10^{-7}	3.98×10^{-7}	3.98×10^{-7}	
	162×162	1.02×10^{-7}	1.02×10^{-7}		
	486×486	1.14×10^{-7}			

The greatest reductions in number of cell updates are achieved by using the smallest subgrids, but at the cost of inducing more computational and memory overhead. Table 7.4 shows the wall-clock speedup (or slowdown) for each run relative (again) to a pure PyClaw run. On static, uniform grids with no subgrids (hence no local time stepping), we again observe a Python callback overhead of up to 40%. In the first column ($\mu = 0$; regular grids with no coarsening) local time stepping is beneficial

Table 7.3: Normalized cell updates for different μ.

Finest Grid size	Subgrid size	Number of levels of coarsening			
		$\mu = 0$	$\mu = 1$	$\mu = 2$	$\mu = 3$
486×486	6×6	0.65	0.20	0.19	
	18×18	0.66	0.40		
	54×54	0.68			
1458×1458	6×6	0.66	0.12	0.10	0.10
	18×18	0.67	0.21	0.20	
	54×54	0.68	0.41		
	162×162	0.71			
4374×4374	18×18	0.67	0.12	0.10	0.10
	54×54	0.68	0.21	0.20	
	162×162	0.69	0.42		
	486×486	0.72			

Table 7.4: Normalized runtime for different μ. The runtime in brackets (left column) is for one PyClaw grid embedded into Peano's root node.

Finest Grid size	Subgrid size	Number of levels of coarsening			
		$\mu = 0$	$\mu = 1$	$\mu = 2$	$\mu = 3$
486×486 (1.05)	6×6	10.60	3.84	3.54	
	18×18	1.96	1.28		
	54×54	1.01			
1458×1458 (1.29)	6×6	17.45	3.96	3.09	3.03
	18×18	2.62	0.97	0.90	
	54×54	1.04	0.67		
	162×162	0.85			
4374×4374 (1.41)	18×18	2.88	0.69	0.56	0.55
	54×54	1.07	0.40	0.38	
	162×162	0.79	0.54		
	486×486	0.82			

only if both the subgrids and the overall grid are reasonably big. Significant speedup (versus PyClaw) is achieved when the finest grid is large (so that dynamic adaptivity and local time stepping have a noticeable effect) and the subgrids are of moderate size (so the overhead is not too large). We note that this problem does not present an ideal case for AMR, since the wave fronts cover a large portion of the domain. Instead, the main benefit here is due to the local time stepping. For problems where a relatively smaller portion of the domain is covered by subgrids of the finest level, more substantial speedup may be expected.

8. Conclusion and Outlook. We have introduced an AMR extension to the Clawpack algorithms, using PyClaw and Peano. While this is an immediately useful feature for PyClaw users, the paper addresses broader, more fundamental questions. It proposes to wrap patch-based PDE solvers into a spacetree formalism. Such a wrapper layer then can autonomously preserve the global data consistency and reuse classical regular Cartesian grid routines unmodified. To switch from regular to adaptive grids

requires minimal effort.

Besides the introduction of this principle—a formalism of an often used pattern [14, 22]—and its rigorous application to AMR, the paper also discusses the realisation of an adaptive local time stepping scheme that comes along with a lower memory footprint than classical schemes and decouples subgrids of the same resolution from each other. Due to this decoupling of the time step size, subgrids on which the numerical solution admits a larger stable and accurate time step size can progress with significantly larger time steps than other regions. This reduces the computing time. To the best of our knowledge, this paper is the first to study the impact of subgrid size choice, and to present a performance model predicting the impact of adaptivity and local time stepping for subgrids.

Future work includes the application of the proposed techniques to real-world problems. Savings in time and memory should allow resolution of real-world problems with a higher accuracy than possible using PyClaw's non-adaptive solvers. The methodologies developed here could also be applied to solvers for other classes of PDEs. Particularly promising is the solution of multiphysics-multiscale problems where multiple PDEs are solved on different scales simultaneously. In contrast to many other tree-based and unstructured grid codes ([9, 12, 23], e.g.), Peano's spacetree holds all resolution levels, and it is trivial to make PeanoClaw embed grids into more nodes than exclusively the leaves. Multiple equations on different spatial scales could then be solved simultaneously.

We see potential with respect to high performance computing. The patch-based callback formalism lends itself naturally to deployment on specialized hardware. The fact that the kernels work on regular data structures makes them well-suited to streaming and SIMD accelerator cards. Such a scheme proposed, for instance, in [14], and is interesting to application developers since concerns related to the particular hardware to be used can also be hidden from the application developer. Furthermore, the patch-based callback formalism naturally suggests a red-black-Gauß-Seidel-type coloring for the patches. Grid updates corresponding to spacetree nodes that do not share a common vertex do not depend on each other (due to (6.4)). We can process those patches simultaneously on a multicore or distributed memory parallel machine [13]. This parallelisation, too, can be hidden from the application developer. Of course, load balancing for solvers with adaptive refinement and local time stepping poses a challenge. Also, efficiency of such parallelisation approaches must be compared to alternative parallelisation approaches integrated directly into PyClaw.

Acknowledgements. This publication is partially based on work supported by Award No. UK-c0020, made by the King Abdullah University of Science and Technology (KAUST). We appreciate the support of the conference [HPC]^3 2012 sponsored by KAUST, and we acknowledge that the technical realisation of the present algorithms was inspired by the work of Neumann in [22]. Special thanks are due to the work done by the Dept. of Computer Science, KU Leuven, in particular Bart Verleye, and the Flanders ExaScience Lab (Intel Labs Europe) with the software Peano. Their requests to enable Peano to assign arrays to spacetree vertices and nodes made this work possible. All software is available at [18, 20] and [26].

REFERENCES

[1] M. Bader, C. Böck, J. Schwaiger, and C. A. Vigh. Dynamically Adaptive Simulations with

Minimal Memory Requirement - Solving the Shallow Water Equations Using Sierpinski Curves. *SIAM Journal of Scientific Computing*, 32(1):212–228, 2010.

[2] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler. Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Softw.*, 38(2):14:1–14:28, January 2012.

[3] J. Bell, M. Berger, J. Saltzman, and M. Welcome. Three-Dimensional Adaptive Mesh Refinement for Hyperbolic Conservation Laws. *SIAM Journal on Scientific Computing*, 15:127–138, 1994.

[4] B. Bergen, T. Gradl, U. Rüde, and F. Hülsemann. A Massively Parallel Multigrid Method for Finite Elements. *Computing in Science & Engineering*, 8(6):56–62, 2006.

[5] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, 1989.

[6] M. J. Berger and R. J. LeVeque. Adaptive Mesh Refinement using Wave-Propagation Algorithms for Hyperbolic Systems. *SIAM Journal Numer. Anal.*, 35:2298–2316, 1998.

[7] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53(3):484 – 512, 1984.

[8] H.-J. Bungartz, M. Mehl, T. Neckel, and T. Weinzierl. The PDE framework Peano applied to fluid dynamics: an efficient implementation of a parallel multiscale fluid dynamics solver on octree-like adaptive Cartesian grids. *Computational Mechanics*, 46(1):103–114, June 2010. published online.

[9] C. Burstedde, L. C. Wilcox, and O. Ghattas. `p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[10] C. E. Castro, M. Käser, and E. F. Toro. Space-time adaptive numerical methods for geophysical applications. *Roy. Soc. Phil. Trans. A*, 367, 2009.

[11] J. Diaz and M. J. Grote. Energy Conserving Explicit Local Time Stepping for Second-Order Wave Equations. *SIAM Journal on Scientific Computing*, 31(3):1985–2014, March 2009.

[12] M. Dumbser, M. Käser, and E. Toro. An Arbitrary High Order Discontinuous Galerkin Method for Elastic Waves on Unstructured Meshes V: Local Time Stepping and p-Adaptivity. *Geophysical Journal International*, 171(2):695–717, 2007.

[13] W. Eckhardt and T. Weinzierl. A Blocking Strategy on Multicore Architectures for Dynamically Adaptive PDE Solvers. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, PPAM 2009*, volume 6068 of *Lecture Notes in Computer Science*, pages 567–575. Springer-Verlag, 2010.

[14] C. Feichtinger, S. Donath, H. Köstler, J. Götz, and U. Rüde. WaLBerla: HPC software design for computational engineering simulations. *Journal of Computational Science*, 2(2):105–112, 2011.

[15] M. J. Gander and L. Halpern. Techniques for Locally Adaptive Timestepping Developed over the Last Two Decades. *Lecture Notes in Computational Science and Engineering*, 2011. submitted to Domain Decomposition Methods in Science and Engineering.

[16] J. E. Dendy Jr. and J. D. Moulton. Black Box Multigrid with coarsening by a factor of three. *Numerical Linear Algebra with Applications*, 17(2–3):577–598, 2010.

[17] D. I. Ketcheson, K. T. Mandli, A. Ahmadia, A. Alghamdi, M. Quezada, M. Parsani, M. G. Knepley, and M. Emmett. PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems. *SIAM Journal on Scientific Computing*, 34(4):C210–C231, 2012.

[18] R. J. LeVeque, M. J. Berger, et al. Clawpack, 2012. www.clawpack.org.

[19] R. J. LeVeque, D. L. George, and M. J. Berger. Tsunami modelling with adaptively refined finite volume methods. *Acta Numerica*, 20:211–289, 2011.

[20] K. T. Mandli, D. I. Ketcheson, et al. Pyclaw software, 2012. numerics.kaust.edu.sa/pyclaw.

[21] O. Meister, K. Rahnema, and M. Bader. A Software Concept for Cache-Efficient Simulation on Dynamically Adaptive Structured Triangular Grids. In K. De Boschhere, E. H. D'Hollander, G. R. Joubert, D. Padua, and F. Peters, editors, *Applications, Tools and Techniques on the Road to Exascale Computing*, volume 22 of *Advances in Parallel Computing*, pages 251–260. ParCo 2012, IOS Press, 2012.

[22] P. Neumann and T. Neckel. A Dynamic Mesh Refinement Technique for Lattice Boltzmann Simulations on Octree-Like Grids. *Computational Mechanics*, 51(2):237–253, 2013.

[23] S. Popinet. Quadtree-adaptive tsunami modelling. *Ocean Dynamics*, 61(9):1261–1285, 2011.

[24] S.-H. Teng. Provably Good Partitioning and Load Balancing Algorithms for Parallel Adaptive N-Body Simulation. *SIAM Journal on Scientific Computing*, 19(2):635–656, 1998.

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

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

framework.org.

[27] 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.