FORESTCLAW: A PARALLEL ALGORITHM FOR PATCH-BASED ADAPTIVE MESH REFINEMENT ON A FOREST OF QUADTREES

DONNA A. CALHOUN* AND CARSTEN BURSTEDDE†

Abstract. We describe a parallel, adaptive, multiblock algorithm for explicit integration of time dependent partial differential equations on two-dimensional Cartesian grids. The grid layout we consider consists of a nested hierarchy of fixed size, non-overlapping, logically Cartesian grids stored as leaves in a quadtree. Dynamic grid refinement and parallel partitioning of the grids is done through the use of the highly scalable quadtree/octree library `p4est`. Because our concept is multiblock, we are able to easily solve on a variety of geometries including the cubed sphere. In this paper, we pay special attention to providing details of the parallel ghost-filling algorithm needed to ensure that both corner and edge ghost regions around each grid hold valid values.

We have implemented this algorithm in the ForestClaw code using single-grid solvers from Clawpack, a software package for solving hyperbolic PDEs using finite volumes methods. We show weak and strong scalability results for scalar advection problems on two-dimensional manifold domains on 1 to 64Ki MPI processes, demonstrating neglible regridding overhead.

Key words. Adaptive mesh refinement, multiblock, finite volume schemes, forest of quadtrees, parallel algorithms

AMS subject classifications. 65M08, 65M50, 68W10, 65Y05

1. Introduction.

The use of spatial adaptivity is widely recognized as an effective way to improve the performance of Cartesian grid methods for partial differential equations (PDEs), and in fact was cited in a recent survey as the main reason for users to adopt a particular code [22]. With the ubiquity of multi-core machines at every level of computing performance, parallel capabilities are expected for codes running on anything from desktop machines to petascale supercomputer architectures. However, building parallel, adaptive capabilities into solvers is a daunting task, and often one which is completely orthogonal to the task of improving the speed and accuracy of single grid solvers for PDEs. While this situation presents difficult challenges with respect to numerical accuracy and parallel performance, we see it as providing a highly motivating opportunity to investigate a modular strategy to adaptive solver development that maximizes the reuse of proven algorithms.

We describe a hybrid approach to adaptive mesh refinement (AMR) that uses the highly scalable quadtree/octree library `p4est` [13,27,28] to manage a dynamic, multiresolution hierarchy of small grids that are distributed in parallel. This hierarchy occupies non-overlapping regions of the computational domain defined by recursively subdividing the domain into quadrants (or octants in 3D). Each region is assigned to an owner process, and the technical issues of parallel mesh management are encapsulated inside the meshing library. Our atomic unit of computation is thus a small uniform grid. Each such grid is processed by a single grid Cartesian solver (e.g. Clawpack, [5]) that mostly encapsulates the solver details. Our software implementation, ForestClaw, provides the central orchestration and coordinates calls between the mesh management library and solver libraries, including those AMR tasks related to parallel neighbor communication, dynamic remeshing (including re-mapping of the solution to newly created meshes) and time stepping.

*Boise State University, Boise ID, USA (corresponding author: `donnacalhoun@boisestate.edu`)

†Institut für Numerische Simulation (INS) and Hausdorff Center for Mathematics (HCM), Universität Bonn, Germany

There are several existing software frameworks for general patch-based parallel AMR, including BOXLIB, AMRCLAW, SAMRAI, PARAMESH, AMROC, UINTAH, CHOMBO and EBCHOMBO [7, 18, 19, 24, 37, 40, 44]. These codes are typically based on finite volume methods for conservation laws on logically Cartesian meshes, and so complex geometries are handled using either mapped grids, or cut-cell approaches, although only EBCHOMBO has extensive support for cut cells. Discussions of survey and experiences in using several of the high level frameworks for block-structured AMR described above can be found in [21, 23].

With the exception of PARAMESH, these software platforms use the original Berger-Oliger and Berger-Collela block-structured mesh approach to dynamic mesh refinement. Such mesh hierarchies consist of nested, overlapping grids of increasingly finer resolution. PARAMESH, which is the mesh management library supporting the FLASH multiphysics code, in contrast, consists of a hierarchy of non-overlapping grids, organized as leaves in a quad- or octree. The approach taken by PARAMESH is thus most closely related to the approach we describe here.

What sets the FORESTCLAW code and project apart from related adaptive mesh methods, notably the current standard approach described by Berger, Oliger, Colella, LeVeque and others for solving conservation laws on Cartesian, finite volume grids [4, 6], is the following.

- Highly scalable quadtree regridding using a the `p4est` parallel mesh backend that has been demonstrated succesfully at the petascale.
- A flexible mapped, multiblock infrastructure for solving on a variety of domains not easily represented by a single Cartesian block. A key infrastructure element includes transformations for handling orientation mis-matches at block boundaries.
- A detailed description of both a serial and parallel algorithm for filling ghost regions of grids stored in the leaves of a quadtree, and:
- Many customizable options, including a package handling infrastructure for binding to multiple solvers.

Additional features that are based on earlier research on Cartesian grid methods include built-in support for computing metric terms needed for solving PDEs on mapped grids that are consistent with second order finite volume schemes.

Even when not using the adaptive refinement features, one can expect to benefit from better cache performance for a uniformly refined domain, more flexible geometric features that the multiblock architecture provides, and a highly flexible and performant parallel partitioning scheme based on space-filling curves.

The main focus of this article will be on the algorithmic details, implemented in the FORESTCLAW code, associated with orchestrating parallel communication between grids for second order finite volume schemes for time dependent hyperbolic PDEs in a two-dimensional quadtree layout. We highlight our main design goals which were chosen to facilitate re-use of code involved in grid communication, and the ease of use for developing new numerical methods. We provide performance results on 1 to 64Ki processes, for the solution of a scalar advection equation, integrated explicitly using a single globally stable time step. Detailed numerical accuracy results, refinement criteria and solutions to more general conservation laws are discussed in this article, but will be presented in a second paper.

2. A quadtree layout. A FORESTCLAW domain consists of a static arrangement of one or more blocks, each of which can be recursively and dynamically subdivided into quadrants. When refinement is requested, a level 0 quadrant, which

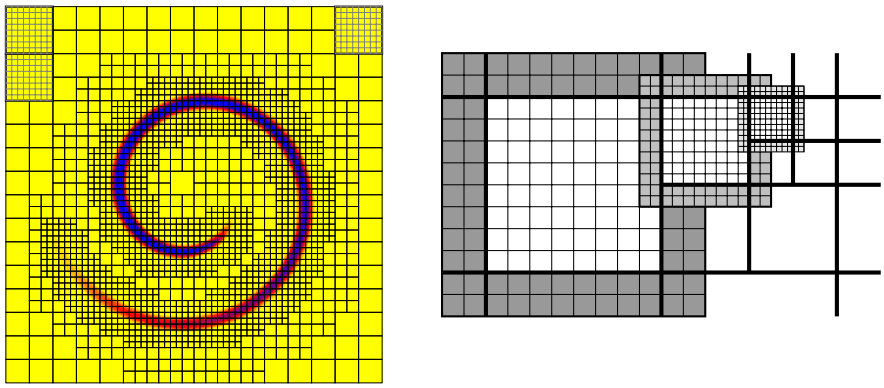

FIG. 2.1. *The left figure shows quadrants in levels 3 through 6 of a adaptive quadtree on a single block. For clarity, we only show grid lines inside the* 8×8 *grids occupying level 3 quadrants (with darker solid borders). The right figure shows three* 8×8 *computational grids, each with a layer of ghost cells, at three adjacent levels. Thick lines indicate quadrant (grid) boundaries.*

occupies the same computational space as its parent block, is partitioned into four equally sized level 1 quadrants. One or more level 1 quadrants can then be partitioned into four level 2 quadrants each, and so on. Coarsening proceeds in reverse. The collection of all quadrants in a block forms a partition of a square computational subdomain, with the length of an edge of a level ℓ quadrant being $2^{-\ell}$ times the length of the level 0 edge. An optional feature of meshes generated using the `p4est` library, and specifically called upon by FORESTCLAW is that meshes can be made *2:1 balanced* [27, 46]. Any two quadrants that share a face or corner will never be more than one level apart. We refer to this arrangement of blocks, each of which is partitioned into one or more quadrants, as a *multiblock quadtree* layout, or simply a *quadtree* layout. Throughout this paper, we will often use the term "patch" to mean a quadrant together with associated numerical data. See Figure 2.1 for a typical single block layout generated by FORESTCLAW.

Each quadrant that makes up the final partitioning of a FORESTCLAW quadtree layout is occupied by a fixed-size, logically Cartesian grid. Each grid has an interior region that fits the quadrant area, made up of typically 8^2, 16^2 or 32^2 grid cells, and one or more layers of ghost cells that extend outside in the two coordinate directions. Solution data on a computational grid is stored in a contiguous array that includes both interior and ghost regions, so that a grid with 8^2 interior cells and two layers of ghost cells stores solution data in a contiguous array of 12^2 mesh cells. The solution at each grid cell stores one or more field variables that make up the numerical solution, as well as any metric dependent data. The interior regions of computational grids do not overlap each other, but the ghost region of one grid will overlap with the interior region of multiple face-adjacent and corner-adjacent neighbors. In FORESTCLAW, values for the interior grid dimensions and number of ghost cell layers are the same for all grids, effectively enforcing a constant 2:1 refinement ratio between grid levels. The *resolution* of a particular grid is determined by the size of the quadrant it occupies, so a grid occupying a level ℓ quadrant has 2^ℓ times the resolution of the same grid in a level 0 quadrant.

When describing numerical schemes, it will be convenient to refer to the border of the interior region (i.e., the quadrant) as the grid *boundary*, even though this boundary

does not enclose the ghost regions belonging to the grid. When the context is clear, the "size" of a grid should be loosely understood to mean the size of the quadrant occupied by that grid, although there will also be occasion to describe a grid using its (fixed) interior dimensions, e.g. a 32×32 grid. It is also informally understood that the use of the term "grid" often refers to the contiguous array of solution values associated with the grid, and not just the geometric metadata needed to describe the grid. In this context, a "coarse grid solution" or a "fine grid solution" is the solution on a coarser or finer grid. In the current version of ForestClaw, we store grids (and solution values) only for those quadrants that make up the final partitioning of the domain. If, during refinement, a coarse quadrant is subdivided into four finer quadrants, the storage for the coarse grid solution and any coarse grid metadata is deleted and storage for a finer grid is allocated in each of the four finer quadrants. See Figure 2.1 for an illustration of grids and quadrants.

There are several advantages to tree-based refinement. One, the numerical analyst developing methods for an adaptive mesh should find it relatively simple to work with the quadtree layout, since quadrant connection patterns appear in only one of three regular arrangements: a neighboring grid is either the same size, twice the size, or one of two half sized grids. Also, it can guaranteed that higher order stencils will have sufficient data from directly adjacent grids and will never need to use data from more than two levels of refinement. Finally, all communication between grids needed for advancing the solution takes place at grid boundaries, reducing the reliance on metadata. From a performance standpoint, tree structures have been extensively studied, and so their performance characteristics in a wide range of scenarios is well understood. The information on neighboring quadrants can be cached and exposed by the meshing library in such a way that they may be located within a tree traversal by $\mathcal{O}(1)$-time lookup functions. Finally, the grids in a quadtree layout can be enumerated to preserve data locality using either Morton ordering (as we do in ForestClaw) or other types of well known space-filling curves [1, 42].

One potential disadvantage of the quadtree layout is that a single quadtree may not be appropriate for a general rectangular domain with a large aspect ratios. In ForestClaw, this difficulty can be overcome in at least two ways. First, one could simply choose fixed size grids with different numbers of grid cells in each of the two coordinate directions. For example, a quadtree in which each leaf contains a 64×16 grid would effectively allow one to grid a 4×1 domain, while maintaining square grid cells. This is done for example in the Racoon code [20]. The downside to this approach is that while individual grid cells have aspect ratios close to 1, the quadrants do not, making it more difficult to efficiently refine around some regions. A second approach, and the approach favored in ForestClaw, is to allow domains to consist of more than one quadtree, or a *forest* of trees. A 4×1 domain is naturally divided into four square blocks, each of which contains a quadtree with square mesh cells. ForestClaw allows for general arrangements of blocks, with the only restriction being that face-adjacent blocks must share a complete face. One surprisingly useful domain is the "brick" domain, an $M \times N$ arrangement of square blocks in an optimized order. In ForestClaw, the brick domain is used for meshing general rectangular domains, the annulus, a spherical coordinate (e.g. latituted/longitude) grid of the mid-latitude region of the earth, and the torus, all with relatively uniform, square mesh cells. Another useful multiblock layout is the cubed sphere grid, widely used in numerical simulations of weather, climate, geodynamics, etc.

A quadtree/octree data structure is commonly used in various kinds of numerical

applications, including the fast multipole method, and computer graphics, making the quadtree a natural choice for interfacing with other libraries. For this reason, it is well suited as a foundation for more general libraries doing mesh refinement [2,43]. A quadrant/octant based approach has been used in other parallel adaptive frameworks, including PARAMESH, NIRVANA, RACOON II, Peano [48], and the Building-Cubes Method [20,30,40,49]. None of these other codes, however, have general multiblock capabilities, or documented performance results for adaptive simulations at petascale. `p4est`, on the other hand, has well-established performance results [8,10,11,41].

The `p4est` algorithms unify the design principles underlying the use of space filling curves for the ordering of elements [25,26,33,39,47], the refinement one or more tree roots into an adaptive forest [3,45], and the use of linear (i.e. leaf-only) octree storage [46]. The terms "leaf" or"quadrant" (used interchangeably) refer to an abstract placeholder for any kind of application data, identified by discrete tree coordinates and their refinement level. The quadrants in `p4est` can be searched and indexed in a random access pattern, which we exploit to assemble $\mathcal{O}(1)$ lookup information on neighbors. We do not make use of compressed encodings of the leaves that would save additional memory at the price of enforcing ordered-only tree traversals [9,48]. The main `p4est` algorithms, including 2:1 balancing and partitioning among the MPI processes, are similar in spirit to collective MPI commands. Applications such as FORESTCLAW can access the quadrant storage and access all required neighborhood information without calling MPI directly.

2.1. Ghost regions. As described above, the interior region of each grid (stored as a leaf of a quadtree) is surrounded by a layer of one or more ghost cells occupying *ghost regions*. All communication between grids is facilitated by copying, averaging or interpolating data from the interior regions of one grid to the ghost regions of a neighboring grid. Since we will want to use unsplit schemes for hyperbolic problems, we also must fill corner ghost regions of each grid with valid data.

We define grid neighbors as those grids occupying either face-adjacent or corner-adjacent quadrants. The coarse grid neighbors of a level ℓ grid are neighboring grids occupying level $\ell - 1$ quadrants, while fine grid neighbors are those neighboring grids occupying level $\ell + 1$ quadrants. We use the term *coarse ghost regions* to refer collectively to those ghost regions which are filled using data copied from the interior of a same-size neighbor or averaged (as described below) from the interior of a fine grid neighbor. *Fine grid ghost regions* are those ghost regions which are filled by interpolating from data in the interior and ghost regions of a neighboring coarse grid. A coarse grid is a grid with coarse grid ghost regions, and a fine grid is a grid with fine grid ghost regions. Depending on the context, a grid can be both a coarse grid and a fine grid. A ghost region is an interior region if it is inside the physical domain, and an exterior ghost region otherwise.

The numerical operations used in filling ghost cells are copying between neighboring grids of the same size, averaging (restricting) data from the interior of the fine grids to ghost cells of a coarse grid, or interpolating (prolongating) from a coarse grid to fine grid ghost cells. At the boundary of the computational domain, we impose physical boundary conditions. For our purposes here, we assume that we have cell-centered data which represents either a cell average value (in the finite volume sense) or cell-centered point-wise values. For the second order schemes we have implemented, these two interpretations are interchangeable.

Two key assumptions in our present algorithm for filling in ghost cells is that copying and averaging only require data from the interior of a neighboring grid cell,

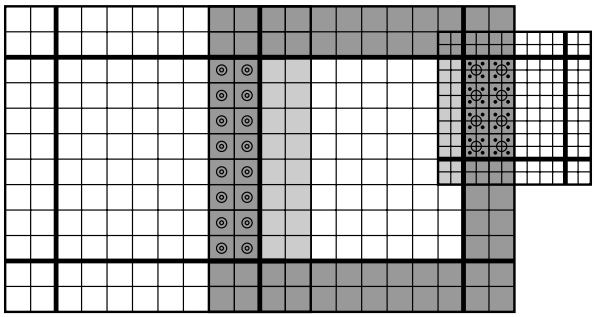

FIG. 2.2. *Dark-shaded coarse ghost regions (of the center grid) are filled by copying from a neighboring same-size grid (left edge) or averaging from a half-size grid (right edge). The open circles are the cell-centered ghost values on the center grid, and smaller black circles are the values on the neighboring grid that are either copied (same-size neighbor) or averaged (finer neighbor).*

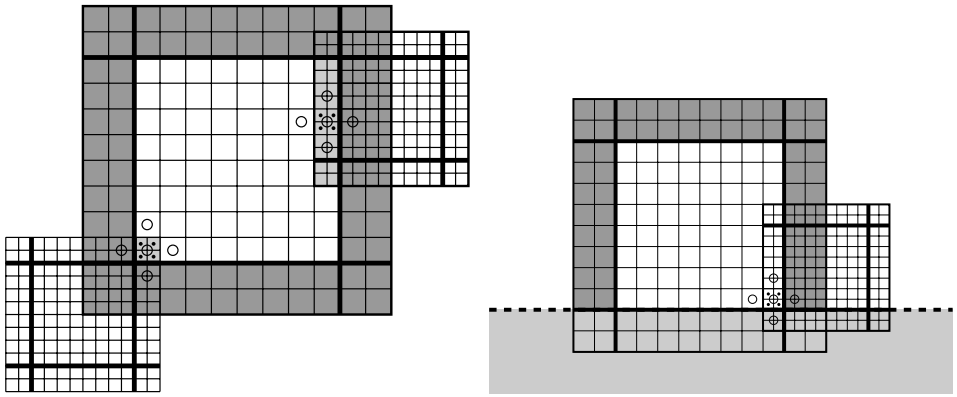

FIG. 2.3. *Coarse grid interpolation stencils used to fill in fine ghost regions. The open circles are the coarse grid values used in the stencil and the filled smaller circles are the fine grid ghost cell values to be filled in. The stencils used in* FORESTCLAW *are applied to a single coarse grid after the coarse grid ghost regions have been filled. The right figure shows fine grid ghost cells at a physical boundary, where the coarse grid interpolation stencil requires valid data from the exterior coarse grid ghost region. This is fabricated according to the boundary condition.*

whereas interpolation will in general require data from both interior and ghost regions. In Figure 2.2, we show typical stencils for averaging and copying from neighboring grids, and in Figure 2.3, we show a typical 5 point stencil used to fill fine grid ghost regions. Because they do not require any data from ghost regions, the coarse grid regions can be filled first, before filling fine grid ghost regions. This ordering of how ghost regions are filled adds some complexity to the serial and parallel ghost filling algorithms, but this additional complexity is more than justified by the greater ease of use obtained with the regular interpolation stencils, especially when going to higher order.

Filling exterior ghost regions is done using standard methods of copying or extending data in some way from the neighboring interior cells, depending on the type of physical boundary condition. The averaging and interpolation stencils we use are shown in Figure 2.3.

2.2. Serial ghost filling algorithm. Assuming that the values in all interior grid cells are valid, either after setting initial conditions or running a time step, it remains to compute correct values in all ghost regions to prepare the following time step. Because our interpolation stencils rely on data in coarse ghost regions, a ghost filling algorithm must ensure that all coarse ghost regions are filled before we fill fine ghost regions. A relatively straightforward algorithm that accomplishes this is presented in Algorithm 1. In this algorithm, the expression "fill coarse ghost regions" means to copy or average data from a neighboring same-size or fine grid neighbor into all edge and corner coarse ghost regions. Conversely, "fill fine ghost regions" refers to using interpolation from coarse neighbors to fill all edge and corner fine ghost regions. To ensure that exterior corner ghost regions have valid data, physical boundary conditions are applied to both edge and corner exterior ghost regions.

The following proposition provides conditions that guarantee that Algorithm 1 fills all edge and corner coarse and fine ghost regions in a quadtree layout with valid data.

PROPOSITION 2.1. *Suppose we have a 2:1 balanced quadtree layout, with grids of fixed size $M \times M$ and m layers of ghost cells each. Let w be the width of the stencil used to interpolate from a coarse grid to a fine ghost region. Assume that the interior regions of all grids contain valid data. Then, if $m \leq M/4$ and $w \leq M/2$, Algorithm 1 is guaranteed to fill in all coarse, fine and exterior ghost regions with valid data.*

To justify this proposition, we only need to demonstrate that an interpolation stencil can never cross more than one *level curve*. Let a *level region* Ω_ℓ (Figure 2.4) be defined as the polygonal region (possibly multiply-connected) containing the interiors of all level ℓ grids. We then define a *level curve* $\Gamma_{\ell+1}$ as the rectilinear curve (or set of curves) separating Ω_ℓ from $\Omega_{\ell+1}$. In a 2:1 balanced quadtree layout, each curve in $\Gamma_{\ell+1}$ is either a simple closed curve (the boundary of a rectilinear polygon), or an open simple rectilinear curve that intersects the physical boundary at each end. Furthermore, if $\ell \neq \ell'$, no curve in Γ_ℓ will intersect a curve in $\Gamma_{\ell'}$ and two curves in Γ_ℓ can intersect only at a corner point. From this, and the other conditions laid out in Proposition 2.1, we conclude that an interpolation stencil cannot cross two level curves from two distinct levels. The practical implication of this is that interpolation stencils will never require data from more than two adjacent levels, and so an algorithm which first fills all coarse ghost regions (traversing the coarse grids in any order), and then fills fine ghost regions (traversing grids in any order), is guaranteed to fill all ghost regions, without breaking any data dependency chains between averaging and interpolation.

2.3. Multiblock indexing. In the low level routines that implement Algorithm 1 (and later, the parallel version) in FORESTCLAW, we explicitly handle several cases of grid arrangements between pairs of neighboring grids.

For pairs of face or corner adjacent grids on the same block, each low-level routine handling the ghost-filling designates a coarse grid ("this" grid) and a same-size or fine "neighbor" grid. We then explicitly handle 20 cases. At each coarse grid face, we have three cases, one for a same-size neighbor, and two for each fine grid neighbor. At each corner, we have two cases, one for a same-size neighbor, and one for a fine grid neighbor. Considering four faces and four corners per patch, we obtain $4\times(3+2) = 20$. There is no advantage in reducing the number of cases below these 20, since doing so would require an expensive remapping of coarse and fine grid data in memory and would lead to code which is hard to read, maintain, or modify. The case of a double-size neighbor is not explicitly handled, since each routine is always written from the

Algorithm 1 Serial algorithm for updating cells in ghost regions on all levels ℓ, $\ell_{\min} \leq \ell \leq \ell_{\max}$. The first application of physical boundary conditions will in general leave data in exterior corner fine ghost regions invalid, requiring a second application of physical boundary conditions after interior fine ghost regions have been filled in an interpolation step.

Require: Solution on interior of all grids contains valid data for given time t.
procedure UPDATE_GHOST
for all levels ℓ, $\ell_{\min} \leq \ell \leq \ell_{\max}$ **do** ▷ Copy and average
Fill all interior coarse ghost regions belonging to level ℓ grids.
end for
for all levels ℓ, $\ell_{\min} \leq \ell \leq \ell_{\max} - 1$ **do** ▷ Apply phys. boundary conditions
Fill exterior coarse ghost regions belong to level ℓ grids.
end for
for all levels ℓ, $\ell_{\min} \leq \ell \leq \ell_{\max} - 1$ **do** ▷ Interpolate
Fill fine ghost regions belonging to level $\ell + 1$ grids.
end for
for all levels ℓ, $\ell_{\min} + 1 \leq \ell \leq \ell_{\max}$ **do** ▷ Apply phys. boundary conditions
Fill exterior ghost regions of level ℓ grids.
end for
end procedure

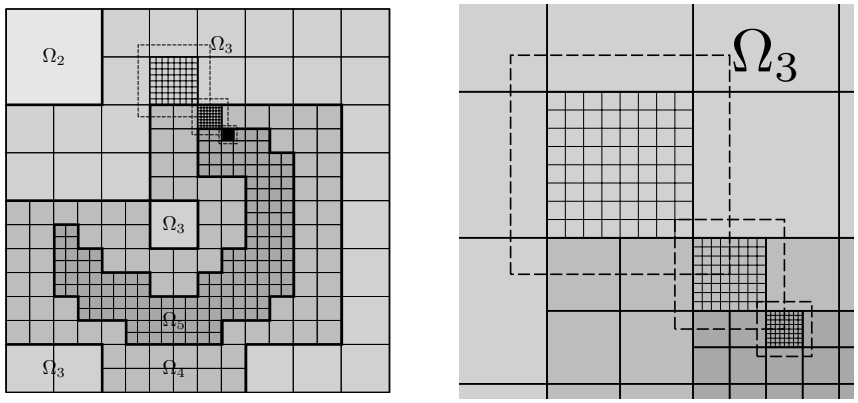

FIG. 2.4. *Typical quadtree layout showing non-overlapping refinement regions Ω_ℓ (left), separated by level curves Γ_ℓ (thick lines). The dashed lines represent the bounds of the ghost regions of selected grids with a zoom-in shown in the right hand picture. By imposing restrictions on the allowable grid size, the number of ghost cell layers, and the width of the interpolation stencil, Algorithm 1 is guaranteed to fill in all corner and edge ghost cell regions with correct values.*

perspective of the coarsest grid.

When two neighboring grids are on different blocks, additional information about the relative orientations of the indices is needed. Holding the coarse grid fixed, the face-adjacent neighboring grid can be rotated through one of four possible positions in the plane, or through two possible positions out of the plane, so that the z-axes (the directions of which are determined from a right hand rule) of the coarse grid and neighboring grid point in opposite directions. Taking into account the two possible positions that a fine grid can have relative to a coarse grid face, the total number of arrangements between a coarse grid and a neighboring grid at one of four coarse

grid faces is 96 (eight positions for each of three types of grids at each of four faces). Similarly, the number of possible configurations for an adjacent corner grid is 32 for both same-size or fine grid neighbors (8 for each same-size corner-adjacent neighbor at each corner, or 8 for each fine grid corner adjacent neighbor).

To avoid this combinatorial explosion of possible grid configurations, ForestClaw makes use of *index transforms* for all corner and face exchanges at grid boundaries, regardless of whether the exchange is between grids on the same block or different blocks. The use of these transforms effectively reduces the complexity in handling multiblock orientations to the 20 cases required by the neighboring grids on the same block.

The index transforms from one index space to the index space of a neighboring same-size grid, possibly at a multiblock boundary, has the general form

$$\mathbf{I}_n = A\mathbf{I}_c + \mathbf{F} \tag{2.1}$$

where A is a 2×2 matrix, $\mathbf{F}$ is a 2×1 vector, and $\mathbf{I}_c$ and $\mathbf{I}_n$ are 2×1 vectors of grid indices (i_c, j_c) and (i_n, j_n). The matrix A encodes the orientation of indices on one patch relative to a second patch, and the vector $\mathbf{F}$ encodes the position of these patches relative to each other. For patches on the same block, the matrix A is the identity matrix, but for patches on different blocks, A will be a diagonal or anti-diagonal matrix whose non-zero entries are 1 or -1. The vector F depends in general on the fixed grid size M.

In what follows, we use $q_c(I_c)$ to indicate a cell-centered value on the coarse grid at index coordinates I_c, defined in coarse grid coordinates. By analogy, the neighboring grid values are indicated using $q_n(I_n)$, where I_n is obtained using (2.1). The transform in (2.1) is provide by `p4est` as a low-level routine, which ForestClaw then uses when copying, averaging and interpolation at patch boundaries, with no distinction made between patches on the same block or on different blocks. With these transformations, the numerical developer can effectively assume all patches have the same index orientation and can essentially implement operations between pairs of patches as if both patches were on the same block.

Copying at patch boundaries. To use multiblock indexing to fill coarse ghost regions via copying, we transform grid cell coordinates $\mathbf{I}_c$ to get index location $\mathbf{I}_n$ on the neighboring same-size grid and then make the assignment $q_c(\mathbf{I}_c) = q_n(\mathbf{I}_n)$.

Averaging and interpolation at patch boundaries. To fill coarse ghost regions via averaging or fine ghost regions via interpolation, we need to map a single coarse grid index I_c to four fine grid indices. We do this by defining four direction vectors on the coarse grid which we use to find four fine grid locations contained within a coarse grid cell. These direction vectors are given by

$$\mathbf{d}_0 = (-1,-1), \quad \mathbf{d}_1 = (1,-1), \quad \mathbf{d}_2 = (-1,1), \quad \mathbf{d}_3 = (1,1). \tag{2.2}$$

The corresponding fine grid locations, in coarse grid coordinates, are then given by

$$\mathbf{I}_c^k = \mathbf{I}_c + \frac{1}{4}\mathbf{d}_k, \qquad k = 0,1,2,3. \tag{2.3}$$

A mapping between coarse and fine grid indices has the general form

$$\mathbf{I}_f = 2A\mathbf{I}_c + \mathbf{F}^f, \tag{2.4}$$

where A encodes index orientations between neighboring coarse and fine grids (as in the same-size transforms), and F^f encodes, in fine grid coordinates, the location of

the fine grid relative to the coarse grid. The four fine grid interpolation points can then be defined as

$$\begin{aligned}\mathbf{I}_f^k &= 2A\left(\mathbf{I}_c + \frac{1}{4}\mathbf{d}_k\right) + \mathbf{F}^f \\ &= 2A\mathbf{I}_c + \frac{1}{2}A\mathbf{d}_k + \mathbf{F}^f, \qquad k = 0, 1, 2, 3.\end{aligned} \tag{2.5}$$

The vector $\mathbf{F}^f$ encodes the location of the center of the coarse grid in fine grid coordinates, and so the entries of $\mathbf{F}^f$ are half-index values. It follows, then, that entries in the vector $\frac{1}{2}A\mathbf{d}_k + \mathbf{F}^f$ are integers, and the final fine grid location $\mathbf{I}_f^k$ will be integer coordinates.

Using the above, we can fill in coarse grid ghost cell values (on a uniform grid) via averaging from a fine grid neighbor as

$$q_c(\mathbf{I}_c) = \frac{1}{4}\sum_{k=0}^{3} q_f(\mathbf{I}_f^k). \tag{2.6}$$

To fill in fine grid ghost cells, we use interpolation stencils that are described entirely in the coarse grid index space, but the index locations of the fine grid ghost cells to be filled in by the interpolation must be obtained via the transformation. The interpolation stencil can be applied as

$$q_f(\mathbf{I}_f^k) = q_c(\mathbf{I}_c) + \frac{h}{4}\widetilde{\nabla} q_c \cdot \mathbf{d}_k \tag{2.7}$$

where $\widetilde{\nabla} q_c$ is an approximation to the gradient of the coarse grid solution and computed using one-sided differences and h is the coarse grid mesh width.

3. Parallel algorithms. Evenly distributing grids to processors using a space-filling curve results in a partitioning of the quadtree layout into logical *processor regions*. See Figure 3.1 for a typical distribution of quadrants to processor regions. Pairs of processor regions are separated by simple rectilinear curves, which we call *processor boundaries*. The *parallel boundary* for a particular processor region is the rectilinear curve made up of all processor boundaries that separate the particular region from all other regions. `p4est` (and by extension, ForestClaw) uses Morton ordering to delineate the space-filling curve. While this is a discontinuous ordering, it can be shown that any processor region defined by this ordering consists of at most two face-connected sub-regions per tree, and so its parallel boundary will consist of at most two simple curves for each tree [12].

Using the single-instruction-multiple-data (SIMD) paradigm underlying the MPI model, we describe our parallel exchange and parallel ghost filling algorithm in terms of a *local* processor region and *remote* processor regions. We refer to *local* grids as those grids whose interiors are in the local processor region. By analogy, remote grids are those grids whose interiors are in remote processor regions. *Local ghost regions* and *remote ghost regions* are ghost regions in local or remote processor regions, respectively. To avoid confusion, we will use terms *local* or *remote* ghost regions for those local or remote regions belonging to local grids only, unless otherwise explicitly stated. Interior regions and ghost regions of a local or remote grid are said to be "on the parallel boundary" if these regions share a corner or face with the parallel boundary.

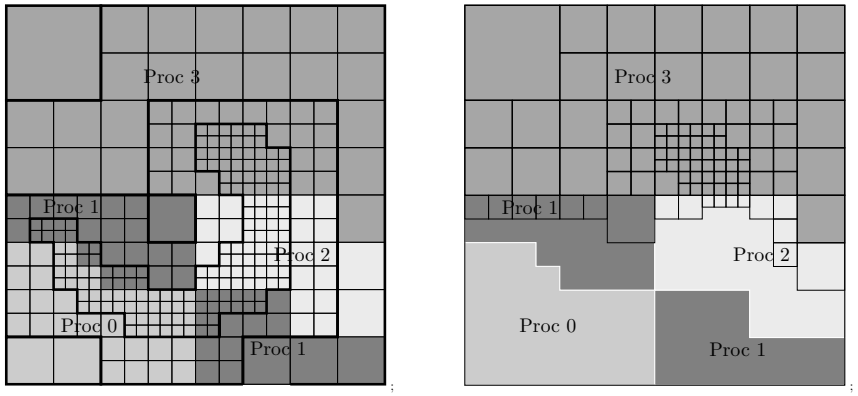

FIG. 3.1. *Shaded regions showing processor regions. The left figure shows patches on all processors, whereas the right figure shows the patches local to processor 3 (thicker lines) and remote ghost patches (thinner lines), sent to processor 3 from processors 1 and 2, and stored as a local copy by processor 3. Only processor region 1 consists of two disjoint subregions.*

If we allow processor boundaries to extend beyond the physical domain in an obvious way, we can also designate exterior ghost regions of local grids as either *local* or *remote*.

3.1. Parallel exchange of ghost cell data. To exchange data across processor boundaries, FORESTCLAW uses the data transfer mechanisms available in `p4est`. First, each processor packs local patches on the parallel boundary into a sender communication buffer. This packing exploits the fact that only the outermost layers of interior cells of a patch need to be sent, while the center region of a patch is never needed by remote processors. Remote patches on the parallel boundary are received by the processor in receiver communication buffers, unpacked accordingly, and stored in a ghost patch array. These local copies of parallel ghost patches are not stored as part of the local tree hierarchy and have limited meta-data, but for the purposes of filling ghost regions, parallel patches can be used just like local neighboring grid patches.

The `p4est` abstract ghost exchange algorithm is based on its internal knowledge of patches on the parallel boundary and can optionally be split into an MPI send phase and an MPI receive phase. This design allows FORESTCLAW to process the local patches not on the parallel boundary between calls to send and receive parallel patches, effectively overlapping communication and computation. Our approach to smoothly grade the refinement described in Section 4.2 also makes use of this abstract `p4est` ghost exchange facility by sending and receiving the target refinement level for each patch. After the target levels for remote patches on the parallel boundary are received, each local patch can compute its target level as the maximum over itself and its direct neighbors without further communication.

3.2. Parallel ghost filling algorithm. Our parallel exchange algorithm needs to manage the sequence of data transfers and mathematical operations between local and remote patches. The challenge in implementing our parallel ghost-filling algorithm is imposing the correct order on the interleaving of steps to fill coarse ghost regions and steps to fill fine ghost regions. Because interpolation stencils may cross multiple parallel boundaries, our algorithm requires multi-way ghost filling between patches

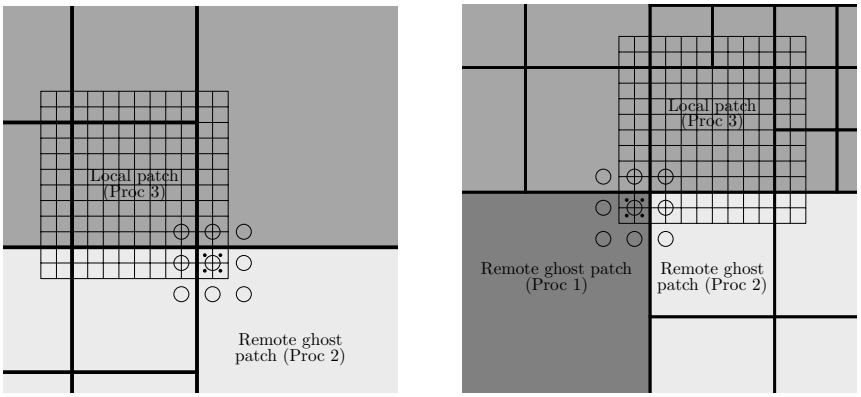

FIG. 3.2. *These figures illustrate the challenge in filling fine corner ghost regions on the parallel boundary. In the left figure, the interpolation stencil needed to fill in the corner fine ghost region on the local patch crosses a single processor boundary and is applied to the remote patch on processor 2. To ensure that the required coarse ghost region on the remote patch has been filled, a pre-processing step on all processors fills coarse ghost regions on the parallel boundary before patches are sent to the remote processors. In the right figure, the interpolation stencil crosses two processor boundaries and the pre-processing step is not sufficient to ensure that the required coarse ghost regions on the remote patch on processor 1 are filled. Therefore, a second ghost filling between locally stored remote patches from processors 1 and 2 is required.*

received from different processors.

In the adaptive setting, parallel boundaries will in general cross boundaries between levels and fine ghost regions belonging to local processors may fall on the parallel boundary. This means that both remote coarse ghost regions belonging to local patches and local coarse ghost regions belonging to remote patches may be required to have valid data before the fine ghost region can be properly filled. For this reason, a pre-processing step on local patches and a post-processing step on remote patches must be carried out before and after parallel communication. It will also be advantageous (even in the uniformly refined case) to hide latency associated with parallel communication. These pre- and post-processing steps and the send and recieve calls split the serial algorithm into three distinct steps. These are labeled **Steps 1, 2 and 3** and detailed in the parallel Algorithm 2.

PROPOSITION 3.1. *Assume we have a 2:1 balanced quadtree layout partitioned to processors using a space-filling curve. Let M be the fixed grid size, let m be the number of ghost layers, and w be the width of the stencil used to interpolate from the coarse grid to fine grid ghost cells. Assume that the interior regions of all grids contain valid data. Then, if $m \leq M/4$ and $w \leq M/2$, Algorithm 2 is guaranteed to fill in all coarse, fine and exterior ghost regions belonging to grids on the local processor with valid data.*

We justify the proposition by describing in detail the three steps of Algorithm 2.

Steps 1 and 2 of our parallel ghost filling algorithm are responsible for filling the coarse grid regions between any two local neighbor grids, and all local fine ghost regions not on the parallel boundary. All local coarse ghost regions can be filled using data from the interior of local patches, or by applying physical boundary conditions. Similarly, all local fine ghost regions not on the parallel boundary can be filled using interpolation stencils that rely only on local coarse grid ghost regions. To overlap communication, Step 1 only fills coarse grid ghost regions on the parallel boundary.

Algorithm 2 Parallel interleaved ghost cell update.

Require: Grids on all levels must be time synchronized
  for all coarse grids **on** the parallel boundary **do** ▷ Step 1
    Fill in local coarse ghost regions
    Apply physical boundary conditions to exterior ghost regions
  end for
  Send local patches at parallel boundary to remote processors
  for all coarse grids **not on** the parallel boundary **do** ▷ Step 2
    Fill in local coarse regions
    Apply physical boundary conditions to exterior ghost regions
    Fill in fine grid neighbors' ghost regions using this grid's interior
  end for
  Receive patches from remote processors
  Fill coarse ghost regions between remote ghost patches from different processors.
  for all coarse grids **on** the parallel boundary **do** ▷ Step 3
    Fill in remote coarse grid ghost regions, using remote grids
    Apply physical boundary conditions to exterior ghost regions
  end for
  for all remote coarse grids **do**
    Fill local coarse ghost regions using local grids.
  end for
  for all grids **on** the parallel boundary **do**
    Fill local coarse ghost regions belonging to remote grids
    Apply physical boundary conditions to exterior ghost regions
  end for
  for all fine grids **on** the parallel boundary **do**
    Fill in remote fine ghost regions, using remote coarse grids.
  end for
  for all grids **on** the parallel boundary **do**
    Apply physical boundary conditions to exterior ghost regions
  end for

Following this step, all grids on the parallel boundary are packed into communication buffers and sent to remote processors. Step 2 then continues by filling all coarse and fine grid regions not on the parallel boundary.

What remains are remote coarse grid regions belonging to local grids and remote and local fine grid ghost regions on the parallel boundary. These remaining ghost regions are filled in Step 3, after the communication step. In the first loop in Step 3, remote coarse grid regions belonging local grids on the parallel boundary are filled by copying or averaging from the locally stored remote parallel patches. To show that local and remote fine grid regions belonging to local grids on the parallel boundary are filled by our algorithm, we have to demonstrate that the stencils involved will have valid data.

In the simplest case, an interpolation stencil needed to fill a remote or local fine grid region on the parallel boundary does not cross the parallel boundary. In this case, the coarse grid ghost data needed for the stencil will have been filled from the averaging step in Step 1 and so the stencil will have valid data.

A slightly more complicated situation occurs when the interpolation stencil crosses

a single parallel boundary. In this situation, the fine grid region to be filled may be either local or remote, but the interpolation stencil will at least partially depend on data from a remote coarse grid ghost region. If the fine grid region is local, then the coarse grid used for the stencil is local, but since it crosses the parallel boundary, the stencil will depend on a remote ghost region belonging to the local coarse grid. These remote ghost regions are filled in the first loop of Step 3. In a second case, the fine grid region is itself a remote region (but belonging to a local grid) and the coarse grid used for the interpolation is itself a remote parallel patch. In this situation, the parallel patch must have valid data in any coarse grid ghost region on the parallel boundary. These regions are filled in the second loop of Step 3. For an illustration of this second case, see the left plot in Figure 3.2.

Finally, the case that requires extra handling is the one in which an interpolation stencil crosses two or more processor boundaries. This situation can arise, for example, when filling corner fine ghost regions that lie in a remote processor region (see the right plot in Figure 3.2). In this case, the remote (but locally stored) parallel ghost patch is not guaranteed to have valid data in all coarse ghost regions on the parallel boundary. If the coarse grid region lies in the local processor region, then this coarse grid ghost region will have been filled in the second loop of Step 3. But if the coarse grid ghost region lies in a third processor's region, then we need a mechanism for filling coarse ghost regions between parallel patches originating from different processors. This is included in Algorithm 2 between Step 2 and Step 3.

4. Dynamic grid adaptation. One of the defining features of ForestClaw, and other adaptive grid codes, is that grids are dynamically adapted to follow solution features of interest. The general refinement strategy in ForestClaw involves applying refinement and coarsening criteria at regular regridding intervals to determine if the solution in a particular quadrant should be replaced with four sibling grids ("refined"), or if the solution on four sibling grids should be replaced by a single parent grid ("coarsened"). Once the refinement and coarsening criteria have been applied, the quadtree mesh is regenerated and the numerical solution is adapted to newly created coarse or fine quadrants. In a parallel setting, this regridding step is followed by a parallel partitioning step that re-distributes grids evenly to processors, correcting potential load imbalances caused by the regridding.

4.1. Dynamic refinement algorithm. The ForestClaw regridding algorithm proceeds in three basic steps. First, each grid in a quadtree layout Q is either tagged for refinement, tagged, along with any sibling grids, for coarsening, or left untagged. A common criterion for tagging cells is a "feature" based refinement, identifying, for example, a sharp jump in the computed solution or a steep gradient. Once tagging has been completed, a quadtree adaptation step creates a new quadtree layout Q'. A 2:1 balancing step, needed to enforce proper nesting of refinement levels, may tag additional grids for refinement, and so coarsening is not always guaranteed to occur based on user-defined coarsening criteria. Grids that are tagged for refinement are guaranteed to be refined.

After the new mesh is generated, we interpolate the solution from coarse grids to the newly created fine grids using the same monotone interpolation scheme we use for filling ghost cells. Sibling grids are averaged to a coarser grid using an averaging stencil. Grids that were neither refined or coarsened are reassigned unchanged from Q to Q'. Once all grids in Q' are populated, grids are re-partitioned to processors to ensure proper load balancing. Again, repartitioning is delegated to the `p4est` library. To support general refinement criteria (examining differences between neighboring

grid cells, for example), the tagging algorithm should run after a call to the ghost filling algorithm. Ghost-filling is required once again on the new and re-partitioned domain Q' to ensure that the adapted solution is well-defined.

The generation of the new adaptive mesh layout Q' and the 2:1 balancing are delegated to the `p4est` library, which operates largely independent of the ForestClaw layer.

4.2. Smooth refinement. When dynamically adapting grid resolution to follow solution features of interest, one wishes to ensure that such features are not too close to coarse-fine boundaries. To provide a buffer region around grids that have been tagged for refinement, ForestClaw can optionally smoothly refine from one region to the next. This effectively adds an additional layer of tagged grids around the finest levels and avoids the situation in which sharp solution features are just barely contained by the finest level grids.

After all grids have been tagged for coarsening or refinement, refinement levels are smoothly graded as follows. Each grid stores its current level and a *target level* which is either equal to the current level (i.e. the grid is not tagged for refinement or coarsening), one greater than the current level, or one level less. Then, for each patch, we compute the maximum of the target level over that patch and all neighboring patches and pass this to the `p4est` adapt/balance routines. The 2:1 balancing algorithm then ensures that neighboring levels will never differ by more than one.

5. Scaling results. We demonstrate our ghost-filling algorithm and parallel communication scheme using the wave propagation algorithms available in Clawpack, a software package for solving hyperbolic problems using high resolution, second order finite volume schemes on logically Cartesian meshes [5, 34–36, 38]. Incorporating the Clawpack Fortran library routines into ForestClaw required only minimal changes to a few of those routines. Both Clawpack 4.6 and Clawpack 5 solvers, along with most Clawpack applications from those packages, are all available as part of ForestClaw.

In the following study, we focus on how the choice of fixed grid size M affects the efficiency of the adaptive algorithms in ForestClaw. The model problems we consider are scalar advection of a tracer field in a square, replicated domain and on a sphere. The scalar advection problem is ideally suited for performance scaling studies because we can easily choose a fixed time step size that remains stable throughout a simulation and across a wide range of resolutions. The Riemann problem for scalar advection has very low arithmetic intensity, so any overhead associated with communication and dynamic regridding cannot be easily hidden by the cost of advancing the solution. Also, because of the low memory requirements of the scalar advection problem, we can run problems that are large enough to maintain appropriate granularity at high processor counts without becoming memory-bound on lower counts.

5.1. Constant velocity in a square, replicated domain. In this first test, we run the scalar advection problem on a sequence of replicated domains designed to provide meaningful weak scaling results for adaptive simulations. Each domain is a multiblock (or "brick") domain consisting of a $2^n \times 2^n$ arrangement of unit blocks, each of which is an adaptive quadtree. The initial tracer field (shown in Figure 5.1) is replicated on each of the blocks in the multiblock domain, and periodic boundary conditions are used at the physical boundaries of the domain. Once the initial tracer field is replicated across the domain, the ForestClaw simulation is oblivious to the replication, and all aspects of the parallel regridding, communication and load

balancing algorithms in ForestClaw are rigorously exercised.

Problem setup. Each unit block in the tracer field on the replicated domain is initialized using the piecewise constant initial field shown in Figure 5.1. The prescribed flow field is the constant velocity $\mathbf{u} = (u, v) = (0.5, 0.5)$, defined using the stream-function $\psi(x, y) = -(x - y)/2$ [14,17]. The periodic boundary conditions imposed on the physical boundary ensure that each block runs the same problem.

For all runs, we fix the CFL number α to 0.64 and adjust the time step for each run to satisfy $\Delta t = \alpha \Delta x$, where Δx is the mesh width for the finest level grids. The number of time steps taken is held fixed so that the final time varies with the resolution of the fixed size grids. For the adaptive runs, we run the simulations for 160 time steps to final times $T = 0.2$, $T = 0.1$ and $T = 0.05$, corresponding to the three different grid resolutions, as described in the next paragraph. The uniform results are run for 20 steps, and resulting timings scaled by 8 to make valid comparisons with the adaptive runs.

We ran three sets of uniform runs and three sets of adaptive runs, corresponding to fixed size grids 8×8, 16×16 and 32×32. For the uniform runs, we set $\ell_{\min} = \ell_{\max} = 7$ so that the effective resolutions on each block in the uniform case are 1024×1024, 2048×2048 and 4096×4096. For the adaptive runs, we set $\ell_{\min} = 4$ and $\ell_{\max} = 7$, corresponding to initial coarse grid resolutions of 128×128, 256×256 and 512×512. The adaptive mesh is dynamically regenerated every $2^{\ell_{\max} - \ell_{\min}} = 2^3 = 8$ times steps. For the uniform runs, we disabled the dynamic regridding and only use the initial mesh created by `p4est`. A grid is tagged for refinement if the difference between its largest value and smallest value exceeds a refinement threshold $\tau_r = 0.25$. Four sibling grids are tagged for coarsening if the difference between the largest and smallest value on each sibling grid does not exceed a coarsening threshold of $\tau_c = 0.001$. The finest level is smoothly graded for all adaptive runs.

Parallel setup. Within each of the six sets of runs described above, we vary the number of processors used and dimensions of the replicated multiblock domain. For example, on the 32×32 runs, we run on replicated domains ranging from a single block to 256×256 blocks, while corresponding MPI process counts vary from 1 to 65,536 (shown in Table 5.1). All of our parallel runs were done on JUQUEEN, the BlueGene/Q system at the Jülich Supercomputing Centre (Forschungszentrum, Jülich, Germany). Each node on JUQUEEN has one 16-core PowerPC A2 processor running at 1.6 GHz with 16 GB RAM. Unless otherwise stated, we ran 32 ranks (MPI processes) on each JUQUEEN node, making 0.5 GB of memory available for each process.

Results. The weak scaling results for the uniform runs (top row of Figure 5.2) show near perfect scaling with close to 100% efficiency on up to 64Ki processes for runs with sufficient granularity. Only the 8×8 run on 64Ki processes with 16 grids per process dips below 80% efficiency. As a point of comparision, Ketcheson, Mandli et al. show 92% efficiency using PyClaw, a massively parallel Python implementation of Clawpack, to solve the Euler equations on up to 64Ki processes on a uniformly refined mesh with 400×400 fixed-sized grids [29].

In the adaptive case, we see from Figure 5.2 that we have close to 90% or better efficiency on up to 4096 processes for all grid sizes, even at very high granularity. For the 32×32 grids, the efficiency on 64Ki only dips below 80% for simulations with 79 grids-per-process and below 60% for 19 grids-per-process. Simulations on the 8×8 grids dips close to or below 60% on 64Ki processes, regardless of granularity, but on 16Ki processes are close to 80% as long as the granularity exceeds roughly 300 grids-

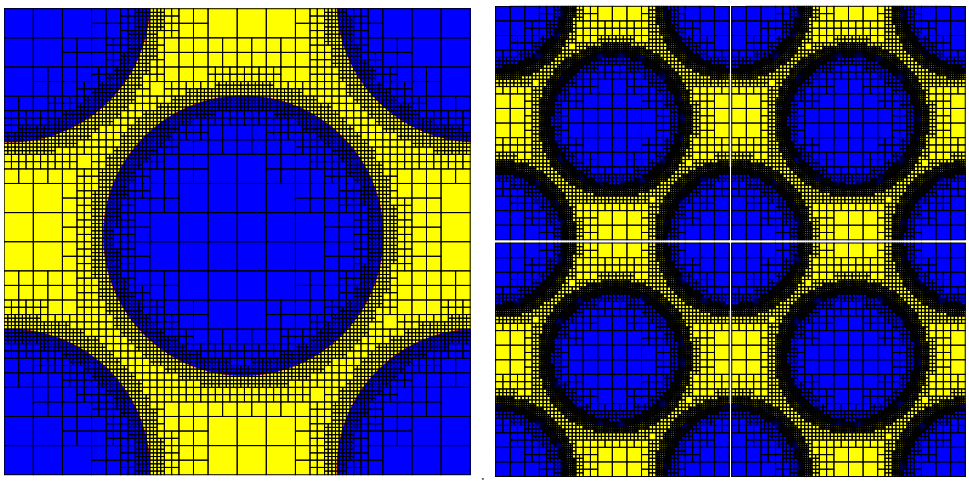

FIG. 5.1. *Initial tracer field for the replicated scalar advection problem (left). The mesh is adapted to the tracer field, which is set to 1 inside of one of five disks of radius 0.3 and 0 outside. We use refinement levels 4 through 7 (grid lines not shown). The domain on the right shows the unit quadtree replicated four times on a* $[0,2] \times [0,2]$ *domain.*

TABLE 5.1

Average number of grids-per-process for the 32×32 *set of runs for the replicated, multiblock scalar advection problem. The leftmost column shows number of* MPI *ranks used for the run and the top row shows the dimensions of the multiblock brick domain used, i.e.* 1×1, 2×2, 4×4 *and so on. Weak scaling results are taken from runs along diagonals, where the work per process remains fixed. We carried out similar sets of runs for* 8×8 *and* 16×16 *fixed size grids, for both adaptive and uniform cases (not shown here).*

Ranks	1	2	4	8	16	32	64	128	256
1	5059	—	—	—	—	—	—	—	—
4	1264	5059	—	—	—	—	—	—	—
16	316	1264	5059	—	—	—	—	—	—
64	79	316	1264	5059	—	—	—	—	—
256	19	79	316	1264	5059	—	—	—	—
1024	—	19	79	316	1264	5059	—	—	—
4096	—	—	19	79	316	1264	5059	—	—
16384	—	—	—	19	79	316	1264	5059	—
65536	—	—	—	—	19	79	316	1264	5059

per-process. The 16×16 runs show close to 80% or better efficiency on up to 16Ki processes for all levels of granularity, whereas the 64Ki runs dip below 80% efficiency, regardless of granularity.

Figure 5.3 shows strong scaling results for all six sets of runs. The data for these results was taken from columns of Table 5.2 (for the 32×32 adaptive run) and similar tables for the other five runs. As with the weak scaling results, the strong scaling results are nearly perfect for the uniform case and show better efficiency for higher resolution fixed size grids in the adaptive case.

In Figure 5.4, we show a breakdown of overhead costs in managing both the uniform and adaptive simulations for the case of 256 grids-per-process in the uniform case, or 300-350 grids-per-process for the adaptive case. In the uniform case, we see that essentially all overhead is in the filling of coarse grid ghost regions via copying

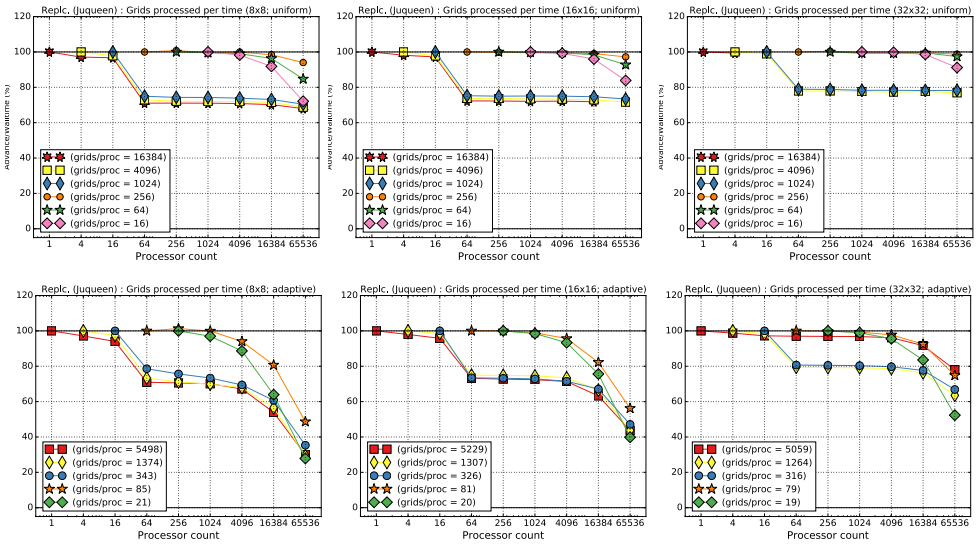

FIG. 5.2. *Weak scaling results for the replicated, multiblock scalar advection problem. The uniform runs are shown in the top row and the adaptive runs on the bottom row. The* 8×8*,* 16×16 *and* 32×32 *runs are shown in the left, middle and right columns, respectively. Each plot shows the efficiency (%) of the particular run, where efficiency is computed as the ratio of the number of grids updated per time for a simulation run on* P *processes to the number of grids updated per time on 1 process. The number of grids-per-process is indicated in the legend and remains constant for each curve. The reduction in efficiency going from 16 processes to 64 processes results from running 1* MPI *process per core on 1,4 and 16 cores to 2 processes per core for 64 and higher process counts, e.g. 32 ranks per node on each JUQUEEN node.*

between grids. Communication costs for the uniform case are negligible and for the 32×32 grids, these communication costs remain essentially flat, even at the highest process counts. For the adaptive 8×8 runs, the adaptive overhead is significant, consuming over 50% of the total time. By contrast, the adaptive 32×32 runs are nearly as efficient, in terms of overhead, as the equivalent uniform runs. Select numerical data from the runs shown in the previous plots are shown in Table 5.2 and Table 5.3. In the results presented here we see a drop in efficiency when going from 16Ki to 64Ki processes. Especially in weak scaling experiments, this is not ideal behavior. We suspect that the overlap of computation and communication we implement is not sufficient at this scale to produce optimal scaling. The precise cause will be investigated using more elaborate profiling.

Finally, we report results from three sets of single-block adaptive runs in which we vary the process count and the maximum level of refinement. This problem is more typical of how one might allocate computational resources, in that one would use additional resources to increase resolution, not run more copies of the same problem. We initialize the single block domain using the same initial conditions and time stepping parameters as in the replicated problem. The range of refinement levels and process counts we chose are shown in Table 5.4.

The results of these single-block runs allow us to obtain interesting insight into the balance between AMR efficiency, granularity and grid size. In Figure 5.5 we show three plots, corresponding to grid sizes 8×8, 16×16 and 32×32. In these plots, we show the relationship between AMR efficiency, measured as fraction of time spent

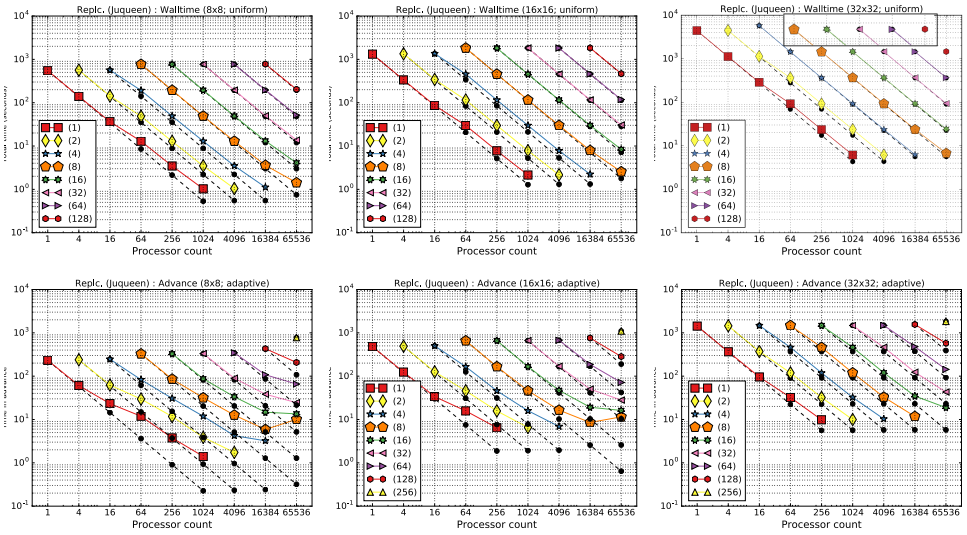

FIG. 5.3. *Strong scaling of replicated uniform problem (top) and the replicated adaptive problem (bottom). The legend labels indicate the number of blocks in each direction in the replicated domain. The values plotted are the wall-clock time for each run. The black dashed line is the ideal scaling, i.e. slope* $= -2$*). The timing results for the lowest granularity simulations in the upper right plot (boxed) could not be computed within allocated time; we estimated them from runs done on higher process counts to complete the picture.*

TABLE 5.2

Wall-clock times (seconds) for the 32×32 *set of runs on the replicated, multiblock scalar advection problem. The leftmost column shows number of* MPI *ranks used for the run, and the top row shows the dimensions of the multiblock brick domain used, i.e.* 1×1*,* 2×2*,* 4×4 *and so on. Times along the diagonals remains relatively fixed, showing good weak scaling results.*

Ranks	1	2	4	8	16	32	64	128	256
1	1430.5	—	—	—	—	—	—	—	—
4	365.6	1450.0	—	—	—	—	—	—	—
16	95.8	371.0	1470.0	—	—	—	—	—	—
64	32.2	119.0	460.0	1470.0	—	—	—	—	—
256	9.85	32.2	119.0	461.0	1480.0	—	—	—	—
1024	—	9.95	32.4	119.0	462.0	1480.0	—	—	—
4096	—	—	10.3	32.9	120.0	465.0	1490.0	—	—
16384	—	—	—	11.8	34.8	123.0	479.0	1560.0	—
65536	—	—	—	—	18.8	43.0	143.0	575.0	1830.0

in AMR tasks and advancing the solution, and granularity. Each plot shows a clear crossover point indicating the minimum granularity needed to ensure that at least 50% of computational time is spent advancing the solution. What the plots clearly show is that this crossover point moves left, towards higher granularity, as the fixed-grid size increases. For 8×8 grids, one needs nearly 1000 grids-per-process before one is spending more time advancing the solution than managing the grids. But for the 32×32 grids, allocating roughly 100 grids-per-process is enough to ensure that over 80% of the total time is spent advancing the solution. Plotting efficiency data from other applications and machine architectures should lead to plots with the same general characteristics as those shown here, and so this novel approach to illustrating

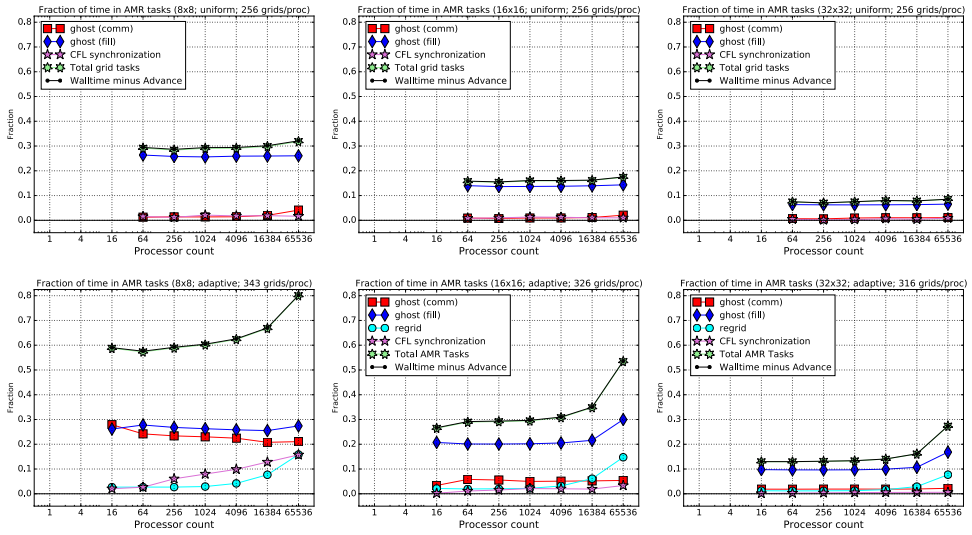

FIG. 5.4. *Fraction of time spent in filling ghost cells, ghost cell communication, CFL synchronization, and (in the adaptive case) regridding on the replicated multiblock scalar advection problem. The top row shows fractions for the uniformly refined simulations, and the bottom row shows fractions for the adaptive simulations. Columns show the* 8×8, 16×16 *and* 32×32 *fixed-size grid runs. Ghost filling tasks include both copying between grids on the same level and averaging and interpolation at coarse/fine interfaces. Regridding tasks include tagging grids, dynamic mesh regeneration, rebuilding newly coarsened or refined grids, and partioning to correct load imbalances. The fractions indicated by the green stars (total time of all AMR tasks shown in the legends), and black dots (wall-clock time minus time spent advancing the solution) are nearly identical, confirming that we have accounted for all significant grid tasks in the indicated legend entries.*

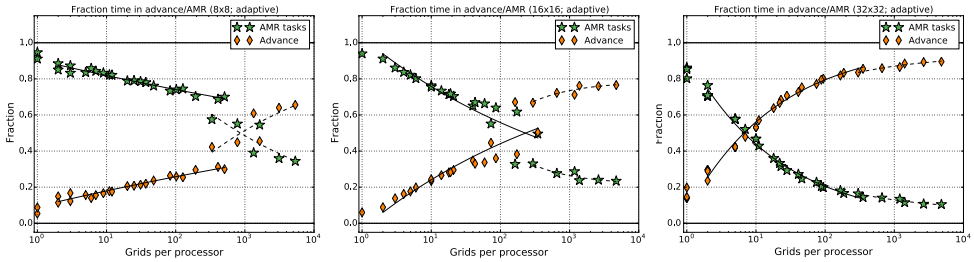

FIG. 5.5. *Plots showing AMR efficiency as a function of granularity (grids-per-process), for the scalar advection problem on a single block. The solid line shows results for process counts of 64 or greater, whereas the dashed line shows results for process counts of 16 or less.*

the dependence of AMR efficiency on granularity and grid size should provide a useful guide in how to allocate computational resources for adaptive simulations using ForestClaw.

5.2. Advection on a sphere. The cubed-sphere grid, shown in Figure 5.6, has become a popular alternative to spherical coordinate grids for solving PDEs on the sphere. This mapping, and many variants, typically have the properties that mesh cells are relatively uniform and they do not suffer from the extreme aspect ratios seen when using spherical coordinates. The cubed-sphere is also an example of a multiblock domain in which grid indices at block boundaries do not generally align

TABLE 5.3

Details of 32×32 run for entries along the top diagonal in Table 5.1 and Table 5.2. The number of grids-per-process for the runs below is fixed at 5059. The rate in the rightmost column is computed as total number of grid cells advanced per time per process. All results were run using 16 ranks per JUQUEEN node.

ForestClaw (32×32; Replicated problem)								
Ranks	Wall	Adv.	(%)	Fill	(%)	Comm.	(%)	Rate
1	1430.5	1292.1	(90.3)	112.5	(7.9)	0.0	(0.0)	5.8×10^5
4	1448.8	1299.1	(89.7)	120.6	(8.3)	2.8	(0.2)	5.7×10^5
16	1471.7	1316.1	(89.4)	123.6	(8.4)	4.9	(0.3)	5.6×10^5
64	1474.9	1317.3	(89.3)	123.7	(8.4)	6.1	(0.4)	5.6×10^5
256	1476.7	1317.5	(89.2)	124.6	(8.4)	5.9	(0.4)	5.6×10^5
1024	1477.8	1316.3	(89.1)	126.1	(8.5)	6.2	(0.4)	5.6×10^5
4096	1487.8	1317.2	(88.5)	134.2	(9.0)	5.2	(0.4)	5.6×10^5
16 384	1561.0	1318.8	(84.5)	204.6	(13.1)	4.4	(0.3)	5.3×10^5
65 536	1831.8	1329.6	(72.6)	449.5	(24.5)	5.4	(0.3)	4.5×10^5

TABLE 5.4

Wall-clock times for the 32×32 adaptive runs on a single block (quadtree) of the replicated problem. The minimum level for each run was fixed at $\ell_{\min} = 4$ and the maximum levels are listed across the top row. The number of grids-per-process for the topmost run in each column is listed in parenthesis in the header for that column. The number of grids-per-process for remaining runs in the column are roughly one fourth of the previous entry.

Ranks	7(4798)	8(2656)	9(1412)	10(186)	11(94)	12(28)
1	1380.0	—	—	—	—	—
4	358.0	1570.0	—	—	—	—
16	93.9	413.0	1740.0	—	—	—
64	31.5	134.0	554.0	—	—	—
256	9.54	37.7	149.0	606.0	—	—
1024	—	12.9	44.5	168.0	644.0	—
4096	—	5.94	17.4	55.3	188.0	450.0

and so is a good test case for the multiblock indexing described in Section 2.3.

Scalar advection on a manifold. To demonstrate the cubed-sphere functionality in FORESTCLAW, we use the tracer transport problem proposed by Lauritzen, Skamarock et al. [31, 32]. The test is intended to assess how well tracer transport schemes can stretch a slotted-disk (see Figure 5.6) and return it to its initial shape. Our goal is to use this example to demonstrate our multiblock mapping capabilities and to assess how metric terms impact the performance of FORESTCLAW.

Two slotted disks are initialized and the velocity field is given in spherical coordinates (ρ, θ) as

$$\Psi(\rho, \theta) = \kappa \sin(\rho - 2\pi t/T)^2 \cos(\theta)^2 \cos(\pi t/T) - 2\pi \sin(\theta)/T_f \tag{5.1}$$

where $\kappa = 2$ and $T_f = 5$. We advect the initial slotted-disk tracer distribution in this velocity field from the initial time to final time $T = 0.5$. For this problem, we test the fixed size 32×32, and, as with the single block example in the previous problem, we vary the refinement levels. A fixed time step on level ℓ is set to $\Delta t_\ell = (2.5 \times 10^{-3})/2^{\ell - \ell_{\min}}$. We run 5 series of runs, from a uniformly refined run at level

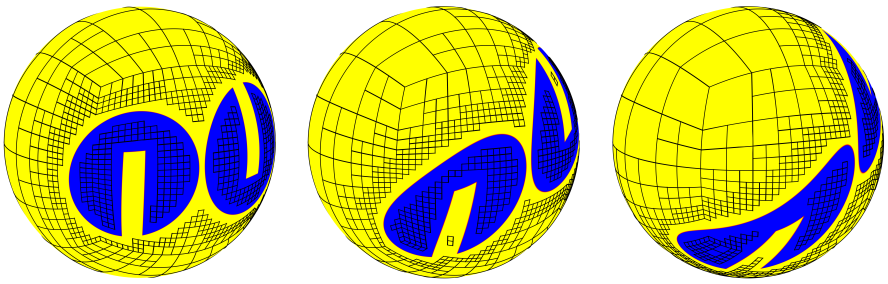

FIG. 5.6. *Three views of the slotted disk problem illustrating flow on a cubed sphere grid for times $t = 0.05$ (left), $t = 0.25$ (middle) and $t = 0.4$ (right). Levels 2–6 are shown (patch borders for level 6 are not shown).*

TABLE 5.5

Wall-clock times and grids-per-process (in parenthesis) for the slotted-disk problem on the cubed-sphere. The header for each column is the maximum level of refinement $\ell_{\max}$ on each block of the six blocks forming the cubed-sphere. The minimum level for all runs is $\ell_{\min} = 2$.

Ranks	2	3	4	5	6
1	274. (96)	1360. (242)	—	—	—
4	72.2 (24)	366. (60)	—	—	—
16	19.2 (6)	111. (15)	508. (38)	—	—
64	—	—	251. (9)	1200. (25)	—
256	—	—	—	489. (6)	1700. (16)

2 (16 32×32 patches per block) to adaptive runs which start at minimum $\ell_{\min} = 2$ and maximum levels varying from 3 to 6. The numerical discretization we use for the sphere is based on CLAWPACK and is described in [15, 16].

Parallel setup. We increase the refinement level and MPI process counts simultaneously. For each of the 5 series of runs, we start with an initial process count and then increase that count by a factor of four as long as the number of grids-per-process exceeds a reasonable threshold (10 or so). In Table 5.5 shows the number of processes for each of 11 runs that we carried out. All runs were done on JUQUEEN.

Results. Figure 5.7 shows the adaptive efficiency as a function of the number of grids-per-process. Adaptive efficiency generally increases with more grids-per-process and reaches over 90% with at least 100 grids-per-process. Even though the runs on the sphere domain have far fewer grids-per-process than on the replicated flat domain, the results in Figure 5.7 have the same characteristics as those seen in Figure 5.5. In Table 5.6, we show the cell processing rate for this problem. This rate is about an order of magnitude smaller than for the flat domain from the previous example. This is due to the extra computational effort required to compute metric terms.

6. Conclusions and future work. In this article, we describe our work in developing a forest-of-quadtrees approach to block-structured adaptive mesh refinement using the algorithms first put forth by Berger and Oliger in 1985. We demonstrate that these ideas can be implemented using the scalable mesh management library `p4est` and hyperbolic solvers from the CLAWPACK library.

A particular focus of this article is on the implementation of the parallel ghost filling algorithm and inter-grid indexing necessary to formulate unsplit finite vol-

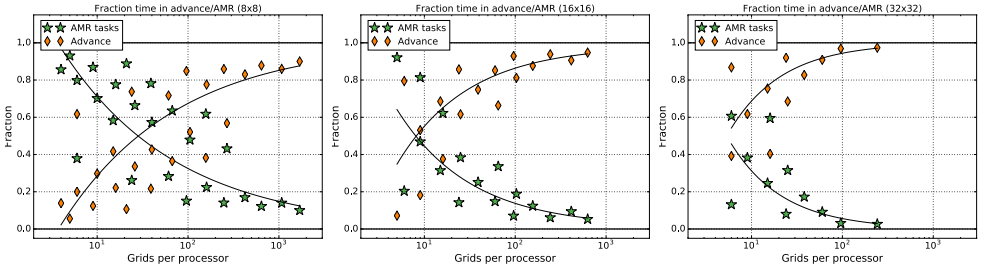

FIG. 5.7. *Adaptive efficiency of the scalar advection problem on a cubed-sphere grid.*

TABLE 5.6

Timing details from slotted-disk problem. Entries are taken along a diagonal from Table 5.5. The rate in the rightmost column is computed as total number of grid cells advanced per time per process. All results were run using 16 ranks per JUQUEEN node.

ForestClaw (32×32; Slotted-disk)										
Ranks	G	$\ell_{\max}$	Wall	Adv. (%)		Fill (%)		Comm. (%)		Rate
1	96	2	274.0	265.0	(96.7)	2.3	(0.8)	0.0	(0.0)	7.2×10^4
4	60	3	366.0	333.0	(91.0)	6.7	(1.8)	16.4	(4.5)	6.8×10^4
16	38	4	508.0	421.0	(82.9)	10.8	(2.1)	53.4	(10.5)	6.2×10^4
64	25	5	1200.0	819.0	(68.3)	20.5	(1.7)	268.0	(22.3)	3.5×10^4
256	16	6	1700.0	688.0	(40.5)	20.7	(1.2)	862.0	(50.7)	3.1×10^4

ume schemes on a multiblock adaptive hierarchy of non-overlapping grids. We add a smooth refinement procedure to preserve moving features of the solution on the finest levels.

We demonstrate our approach numerically by solving a scalar advection problem on 1 to 64Ki MPI processes with good parallel scalability. We also solve an advection problem on the cubed sphere that is composed of multiple blocks. In addition, we looked carefully at how AMR efficiency depends on the granularity (grids-per-process) and fix-grid sizes. We find that 32×32 grids strike a favorable compromise between the flexible refinement offered by small grid sizes and high arithmetic intensity offered by larger sizes.

While FORESTCLAW supports multirate (locally adaptive) time stepping, we only demonstrate global time stepping here. A follow-up article will describe the multirate time stepping capabilities currently available in FORESTCLAW, report on detailed verification studies, comparisons with other adaptive codes, and results from solving more complex hyperbolic systems including the shallow-water and Euler equations.

7. Acknowledgements. Donna Calhoun would like to acknowledge the Isaac Newton Institute program "Multiscale Numerics for the Ocean and Atmosphere" for its support of much of this work during the fall of 2012 and the National Science Foundation (NSF DMS-1419108). Carsten Burstedde is supported by the Hausdorff Center for Mathematics (HCM) at Bonn University funded by the German Research Foundation (DFG).

The authors gratefully acknowledge the Gauß Centre for Supercomputing (GCS) for providing computing time through the John von Neumann Institute for Com-

puting (NIC) on the GCS share of the supercomputer JUQUEEN at Jülich Supercomputing Centre (JSC). GCS is the alliance of the three national supercomputing centres HLRS (Universität Stuttgart), JSC (Forschungszentrum Jülich), and LRZ (Bayerische Akademie der Wissenschaften), funded by the German Federal Ministry of Education and Research (BMBF) and the German State Ministries for Research of Baden-Württemberg (MWK), Bayern (StMWFK) and Nordrhein-Westfalen (MIWF).

REFERENCES

[1] M. Bader, *Space-Filling Curves: An Introduction with Applications in Scientific Computing*, Texts in Computational Science and Engineering, Springer, 2012.

[2] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, *Algorithms and data structures for massively parallel generic adaptive finite element codes*, ACM Transactions on Mathematical Software, 38 (2011), pp. 14:1–14:28.

[3] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.II – a general-purpose object-oriented finite element library*, ACM Transactions on Mathematical Software, 33 (2007), p. 24.

[4] M. J. Berger and P. Colella, *Local adaptive mesh refinement for shock hydrodynamics*, J. Comput. Phys., 82 (1989), pp. 64–84.

[5] M. J. Berger and R. J. LeVeque, *Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems*, SIAM J. Num. Anal., 35 (1998), pp. 2298–2316.

[6] M. J. Berger and J. Oliger, *Adaptive mesh refinement for hyperbolic partial differential equations*, J. Comput. Phys., 53 (1984), pp. 484–512.

[7] *BoxLib*. `https://ccse.lbl.gov/BoxLib/` (last accessed March 8, 2017).

[8] T. Bui-Thanh, C. Burstedde, O. Ghattas, J. Martin, G. Stadler, and L. C. Wilcox, *Extreme-scale UQ for Bayesian inverse problems governed by PDEs*, in SC12: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2012.

[9] H.-J. Bungartz, M. Mehl, and T. Weinzierl, *A parallel adaptive Cartesian PDE solver using space–filling curves*, Euro-Par 2006 Parallel Processing, (2006), pp. 1064–1074.

[10] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. C. Wilcox, *Extreme-scale AMR*, in SC10: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM/IEEE, 2010.

[11] C. Burstedde, O. Ghattas, M. Gurnis, E. Tan, T. Tu, G. Stadler, L. C. Wilcox, and S. Zhong, *Scalable adaptive mantle convection simulation on petascale supercomputers*, in SC08: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM/IEEE, 2008.

[12] C. Burstedde and T. Isaac, *Morton curve segments produce no more than two distinct face-connected subdomains*. http://arxiv.org/abs/1505.05055, 2015.

[13] 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 (2011), pp. 1103–1133.

[14] D. Calhoun, *A Cartesian Grid Method for Solving the Two-Dimensional Streamfunction-Vorticity Equations in Irregular Regions*, J. Comput. Phys., 176 (2002), pp. 231 – 275.

[15] D. Calhoun and C. Helzel, *A Finite Volume Method for Solving Parabolic Equations on Logically Cartesian Curved Surface Meshes*, SIAM J. Sci. Comput., 31 (2009), pp. 4066–4099.

[16] D. Calhoun, C. Helzel, and R. J. LeVeque, *Logically rectangular grids and finite volume methods for PDEs in circular and spherical domains*, SIAM Review, 50 (2008), pp. 723–752.

[17] D. Calhoun and R. J. LeVeque, *Solving the advection-diffusion equation in irregular geometries*, J. Comput. Phys., 156 (2000), pp. 1–38.

[18] *Chombo – Software for Adaptive Solutions of Partial Differential Equations*. `http://seesar.lbl.gov/anag/chombo` (last accessed March 8, 2017).

[19] R. Deiterding, *AMROC Fluid-solver Framework*. `http://www.rdeiterding.website/pub/AMROC/code/amroc/doc/html/amr/index.html`(last accessed on March 8, 2017).

[20] J. Dreher and R. Grauer, *Racoon: A parallel mesh-adaptive framework for hyperbolic conservation laws*, Parallel Computing, 31 (2005), pp. 913–932.

[21] A. Dubey, A. Almgren, J. B. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Läffler, B. OâShea, E. Schnetter, B. V. Straalen, and K. Weide, *A survey of high level frameworks in block-structured adaptive mesh refinement*

packages, Journal of Parallel and Distributed Computing, (2014), pp. –.

[22] A. Dubey, K. Antypas, M. K. Ganapathy, L. B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide, *Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code*, Parallel Computing, 35 (2009), pp. 512 – 522.

[23] A. Dubey and B. V. Straalen, *Experiences from Software Engineering of Large Scale AMR Multiphysics Code Frameworks*, Journal of Open Research Software, 2 (2014), pp. 1–5.

[24] *EBChombo – Embedded Boundary Infrastructure.* `https://commons.lbl.gov/display/chombo/Embedded+Boundary+approach` (last accessed March 8, 2017).

[25] M. Griebel and G. W. Zumbusch, *Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves*, Parallel Computing, 25 (1999), pp. 827–843.

[26] D. Hilbert, *Über die stetige Abbildung einer Linie auf ein Flächenstück*, Mathematische Annalen, 38 (1891), pp. 459–460.

[27] T. Isaac, C. Burstedde, and O. Ghattas, *Low-cost parallel algorithms for 2:1 octree balance*, in Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium, IEEE, 2012. `http://dx.doi.org/10.1109/IPDPS.2012.47`.

[28] T. Isaac, C. Burstedde, L. C. Wilcox, and O. Ghattas, *Recursive algorithms for distributed forests of octrees*, SIAM Journal on Scientific Computing, 37 (2015), pp. C497–C531.

[29] D. I. Ketcheson, K. Mandli, A. J. Ahmadia, A. Alghamdi, M. Q. de Luna, M. Parsani, M. G. Knepley, and M. Emmett, *PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems*, SIAM J. Sci. Comput., 34 (2012), pp. C210–C231.

[30] K. Komatsu, T. Soga, R. Egawa, H. Takizawa, H. Kobayashi, S. Takahashi, D. Sasaki, and K. Nakahashi, *Parallel processing of the Building-Cube Method on a GPU platform*, Computers and Fluids, 45 (2011), pp. 122–128.

[31] P. H. Lauritzen, W. C. Skamarock, M. J. Prather, and M. A. Taylor, *A standard test case suite for two-dimensional linear transport on the sphere*, Geosci. Model Dev., 5 (2012), pp. 887–901.

[32] P. H. Lauritzen, P. A. Ullrich, C. Jablonowski, P. A. Bosler, D. Calhoun, A. J. Conley, T. Enomoto, L. Dong, S. Dubey, O. Guba, A. B. Hansen, E. Kaas, J. Kent, J.-F. Lamarque, M. J. Prather, D. Reinert, V. V. Shashkin, W. C. Skamarock, B. Sorensen, M. A. Taylor, and M. A. Tolstykh, *A standard test case suite for two-dimensional linear transport on the sphere: results from a collection of state-of-the-art schemes*, Geosci. Model Dev., 7 (2014), pp. 105–145.

[33] H. L. Lebesgue, *Leçons sur l'intégration et la recherche des fonctions primitives*, Gauthier-Villars, 1904.

[34] R. J. LeVeque, *High-Resolution Conservative Algorithms for Advection in Incompressible Flow*, SIAM J. Num. Anal., 33 (1996), pp. 627–665.

[35] R. J. LeVeque, *Wave propagation algorithms for multidimensional hyperbolic systems*, J. Comput. Phys., 131 (1997), pp. 327–353.

[36] R. J. LeVeque, *Finite volume methods for hyperbolic problems*, Cambridge University Press, 2002.

[37] R. J. LeVeque and M. J. Berger, AMRClaw.

[38] K. T. Mandli, A. J. Ahmadia, M. J. Berger, D. A. Calhoun, D. L. George, Y. Hadjimichael, D. I. Ketcheson, G. I. Lemoine, and R. J. LeVeque, *Clawpack: Building an open source ecosystem for solving hyperbolic PDEs*, PeerJ Preprint, (2016).

[39] G. M. Morton, *A computer oriented geodetic data base; and a new technique in file sequencing*, tech. rep., IBM Ltd., 1966.

[40] *PARAMESH.* `https://opensource.gsfc.nasa.gov/projects/paramesh/index.php` (last accessed March 8, 2017).

[41] J. Rudi, A. C. I. Malossi, T. Isaac, G. Stadler, M. Gurnis, P. W. J. Staar, Y. Ineichen, C. Bekas, A. Curioni, and O. Ghattas, *An extreme-scale implicit solver for complex pdes: highly heterogeneous flow in earth's mantle*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2015, p. 5.

[42] H. Samet, *Foundations of multidimensional and metric data structures*, Elsevier/Morgan Kaufmann, 2006.

[43] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros, *Dendro: Parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees*, in SC'08: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM/IEEE, 2008.

[44] *SAMRAI: Structured Adaptive Mesh Refinement Application Infrastructure.* `http://computation.llnl.gov/projects/samrai` (last accessed March 8, 2017).

[45] J. R. Stewart and H. C. Edwards, *A framework approach for developing parallel adaptive*

multiphysics applications, Finite Elements in Analysis and Design, 40 (2004), pp. 1599–1617.

[46] H. Sundar, R. Sampath, and G. Biros, *Bottom-up construction and 2:1 balance refinement of linear octrees in parallel*, SIAM Journal on Scientific Computing, 30 (2008), pp. 2675–2708.

[47] T. Tu, D. R. O'Hallaron, and O. Ghattas, *Scalable parallel octree meshing for terascale applications*, in SC '05: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM/IEEE, 2005.

[48] T. Weinzierl and M. Mehl, *Peano—a traversal and storage scheme for octree-like adaptive Cartesian multiscale grids*, SIAM Journal on Scientific Computing, 33 (2011), pp. 2732–2760.

[49] U. Ziegler, *Block-Structured Adaptive Mesh Refinement on Curvilinear-Orthogonal Grids*, SIAM J. Sci. Comput., 34 (2012), pp. C102–C121.