Towards Adaptive Mesh PDE Simulations on Petascale Computers

Carsten Burstedde*, Omar Ghattas*†, Georg Stadler*, Tiankai Tu*, Lucas C. Wilcox*

*Institute for Computational Engineering & Sciences
†Jackson School of Geosciences and Department of Mechanical Engineering
The University of Texas at Austin, Austin, Texas, USA.

1 Introduction

The advent of the age of petascale computing brings unprecedented opportunities for breakthroughs in scientific understanding and engineering innovation. However, the raw performance made available by petascale systems is by itself not sufficient to solve many challenging modeling and simulation problems. For example, the complexity of solving evolutionary partial differential equations scales as n^4 at best, where n is the number of mesh points in each spatial direction.[1] Thus, the three-orders-of-magnitude improvement in peak speed of supercomputers over the past dozen years has meant just a factor of 5.6 improvement in spatio-temporal resolution—not even three successive refinements of mesh size. For many problems of scientific and engineering interest, there is a desire to increase resolution of current simulations by several orders of magnitude. As just one example, current long-range climate models operate at $\mathcal{O}(300)$ km resolution; yet, $\mathcal{O}(10)$ km grid spacing is desirable to obtain predictions of surface temperature and precipitation in sufficient detail to analyze the regional and local implications of climate change, and to resolve oceanic mesoscale eddies [9]. Thus, although supercomputing performance has outpaced Moore's Law over the past several decades due to increased concurrency [9], the curse of dimensionality imposes much slower scientific returns; e.g. improvement in mesh resolution grows at best with the one-fourth power of peak performance in the case of evolutionary PDEs.

The work requirements of scientific simulations typically scale as n^α. The power α can be reduced through the use of optimal solvers such as multigrid for PDEs and fast multipole for integral equations and N-body problems. Once α has been reduced as much as possible, further reductions in work can be achieved only by reducing n itself. This can be achieved in two ways: through the use of adaptive mesh refinement/coarsening (AMR) strategies, and by invoking higher order approximations (in space and time). The former place mesh points only where needed to resolve solution features, while the latter reduce the number of necessary mesh points to achieve a given accuracy.

Fortunately, many problems have local multiscale character, i.e. resolution is needed only in localized (possibly dynamically evolving) regions, such as near fronts, discontinuities, material interfaces, reentrant corners, boundary and interior layers, and so on. In this case, AMR methods can deliver orders-of-magnitude reductions in number of mesh points, e.g., [1, 2, 5, 10]. Unfortunately, AMR methods can also impose significant overhead, in particular on highly parallel computing systems, due to their need for frequent re-adaptation and load-balancing of the mesh over the course of the simulation. Because of the complex data structures and communication patterns and frequent communication and data redistribution, scaling dynamic AMR to tens of thousands of processors has long been considered a challenge.[2]

AMR methods generally fall into two categories, structured (SAMR) and unstructured (UAMR) (see the review in [11] and the references therein). SAMR methods represent the PDE solution on a composite of hierarchical, adaptively-generated, (logically-) rectangular, structured grids. This affords reuse of structured grid sequential codes by nesting the regular grids during refinement. Moreover, higher performance can be achieved due to the regular grid structure. SAMR methods maintain consistency and accuracy of the numerical approximation by carefully managing interactions and interpolations between the nested grids, which makes high-order-accurate methods significantly more difficult to implement than in the single-grid case. The resulting communications, load balancing, and grid interactions present challenges in scaling to $\mathcal{O}(10^4)$ processors. Exemplary SAMR implementations that have scaled to several thousand processors include Chombo [8], PARAMESH [7], and SAMRAI [21].

[1] Optimal solvers require $\mathcal{O}(n^3)$ work per time step, and time accurate integration often implies $\mathcal{O}(n)$ time steps.

[2] For example, the 1997 Petaflops Algorithms Workshop [3] assessed the prospects of a number of high performance computing algorithms scaling to petaflops computers. Algorithms were classified according to Class 1 (scalable with appropriate effort), Class 2 (scalable provided significant research challenges are overcome), and Class 3 (possessing major impediments to scalability). The development of dynamic grid methods—including mesh generation, mesh adaptation and load balancing–were designated Class 2, and thus pose a significant research challenge. In contrast, static-grid PDE solvers were classified as Class 1.

In contrast to SAMR, UAMR methods typically employ a single (often conforming) mesh that is locally adapted by splitting and/or aggregating (often tetrahedral) elements. As such, high-order accuracy is achieved naturally with, e.g., high order finite elements, and the unstructured nature of the tetrahedra permits boundary conformance. Furthermore, the conforming property of the mesh eliminates the need for interpolations between grid levels of SAMR methods. The challenge with UAMR methods is to maintain element quality while adapting the mesh, which is particularly difficult to do in parallel due to the need to coordinate the coarsening/refinement between processors. As in SAMR, dynamic load-balancing, significant communication, and complex data structures must be overcome. Exemplary UAMR implementations include Pyramid [16] and the work of the RPI group [12].

In this paper, we present ALPS (**A**daptive **L**arge-scale **P**arallel **S**imulations), a library for parallel octree-based dynamic mesh adaptivity and redistribution that is designed to scale to hundreds of thousands of compute cores. ALPS uses parallel octree-based non-overlapping hexahedral finite element meshes and dynamic load balancing based on space-filling curves. Flexibility of the mesh is attained by associating a hexahedral finite element to each octree leaf. This approach combines the advantages of UAMR and SAMR methods. Like UAMR, a conforming approximation is achieved, though here the mesh itself is not conforming; instead, algebraic constraints on hanging nodes impose continuity of the solution field across coarse-to-fine element transitions. Thus, high-order approximations (with fixed polynomial degree) are naturally accommodated. Communication is significantly reduced compared to SAMR, since interpolations from coarse to fine elements are limited to face/edge information and occur *just once each mesh adaptation step*, since a single mesh is used to represent the solution at each time step. On the other hand, like SAMR, mesh quality is not an issue, since refinement and coarsening result from straightforward splitting or merging of hexahedral elements.

We assess the performance of ALPS for adaptive solution of dynamic advection-dominated transport problems with sharp fronts. Using TACC's Ranger system, we demonstrate excellent weak and strong scalability on up to 32K cores and 4.3 billion elements. All of the timings and performance evaluations presented in this paper correspond to *entire end-to-end simulations*, including mesh initialization, coarsening, refinement, octree 2:1 balancing, octree partitioning, field transfer, hanging node constraint enforcement, and explicit PDE solution. The results show that, with careful algorithm design and implementation, the cost of dynamic adaptation can be made small relative to that of PDE solution, even for meshes (and partitions) that change drastically over time. Scalability to $\mathcal{O}(10^4)$ cores follows.

In the remainder of this paper, we provide an overview of our approach and present performance studies. Section 2 describes the essential AMR components of ALPS, while Section 3 provides a discussion of the strategy employed for dynamic adaptivity. Assessments of load balance and weak and strong scalability are provided in Section 4.

2 Parallel Octree-Based Adaptive Meshing

In this section we describe the essential components of ALPS. The design of our library supports many mesh-based PDE discretization schemes, such as low- and high-order variants of finite element, finite volume, spectral element, and discontinuous Galerkin methods, though only finite element methods on trilinear hexahedral elements are currently implemented. We build on prior approaches to parallel octree mesh generation [19, 20], and extend them to accommodate dynamic solution-adaptive refinement and coarsening. This requires separating the octree from the mesh data structures. Specifically, adaptation and partitioning of the mesh are handled through the octree structure, and a distinct mesh is generated from the octree every time the mesh changes.

Nonconforming hexahedral meshes of a given rectangular domain are generated for use with a trilinear finite element discretization, and solution fields are made conforming via algebraic continuity constraints on hanging nodes as mentioned in the Introduction. These constraints are eliminated at the element level, so variables at the hanging nodes are no longer degrees of freedom for the solver. We maintain a global 2-to-1 balance condition, i.e., the edge lengths of face- and edge-neighboring elements may differ by at most a factor of 2. This ensures smooth gradations in mesh size, and simplifies the incorporation of algebraic constraints. Octree-based refinement/coarsening of hexahedral finite element meshes with hanging node constraints has been employed in such parallel finite element libraries as deal.II [4], libMesh [14], *hp*3d [10], and AFEAPI [15], and have been demonstrated to scale to well to hundreds of processors. Here, our focus is on new distributed data structures, parallel algorithms, and implementations that scale to $\mathcal{O}(10^4)$ cores. These are discussed in the remainder of this section.

2.1 Octrees and Space-Filling Curves

All coarsening and refinement information is maintained within an octree data structure, in which there is a one-to-one correspondence between the leaves of the octree and the hexahedral elements of the mesh (see Figure 1). The root of the octree represents an octant of the size of the computational domain. The leaves of the octree represent the elements which are present in the current mesh. The parents of these leaves are used to determine the relationships between the leaves. When an element is refined, it is split into eight equal-sized child elements. This is represented in the octree by adding eight children to the leaf

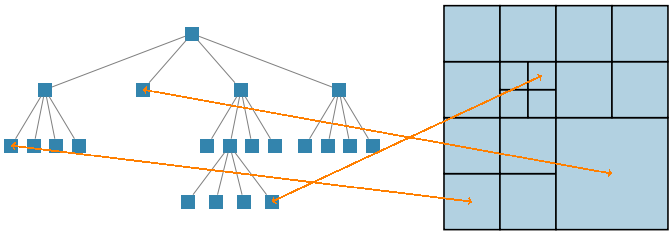

Figure 1: Illustration of the distinct octree and mesh data structures used in ALPS. The data structures are linked logically by a 1-to-1 correspondence between leaves and elements. (A quadtree is show for display purposes.)

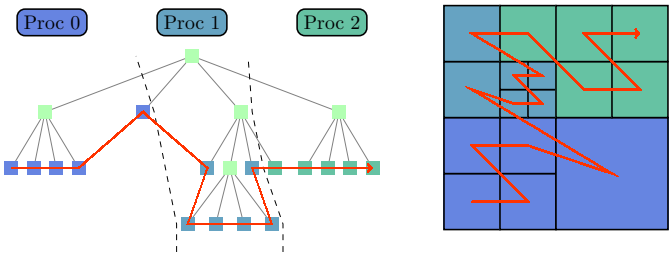

Figure 2: A pre-order traversal of the leaves of the octree in the sequence of triples (z, y, x) creates a space-filling curve in z-order. This imposes a total ordering of the mesh elements, also known as Morton ordering. A load-balanced partition of the octree is determined by partitioning the space-filling curve into segments of equal length. The globally shared information required for this operation amounts to one long integer per core.

octant representing the element being divided. A coarsening operation amounts to removing all children with a common parent. The operations defined on the octree and the mesh are detailed below.

Most of the AMR functions in ALPS operate on the octree from which the mesh is generated. Since we target large parallel systems, we cannot store the full octree on each core. Thus, the tree is partitioned across cores. As we will see below, cores must be able to determine which core owns a given leaf octant. To this end we rely on a space-filling curve, which provides a globally unique linear ordering of all leaves. As a direct consequence, each core stores only the range of leaves each other core owns. This can be determined by an `MPI_Allgather` call on an array of long integers with a length equal to the number of cores. This is the only global information that is required to be stored. We use the Morton ordering as the specific choice of space-filling curve. It has the property that nearby leaves tend to correspond to nearby elements given by the pre-order traversal of the octree, as illustrated in Figure 2.

The basic operations needed for mesh generation and adaptation require each core to find the leaf in the octree corresponding to a given element. If the given element does not exist on the local core, the remote core that owns the element must be determined. This can be done efficiently given the linear order of the octree; see [20] for details. The inverse of this operation, determining the element corresponding to a given leaf, can be made efficient as well.

2.2 MESH GENERATION AND ADAPTATION

The generation of the mesh comprises several distinct steps. There are two scenarios in which a mesh is generated: the first is the initial generation of the mesh, and the second is the generation of a mesh from an adapted octree. As we will see, the adaptation of the mesh in conjunction with the transfer of data fields requires an intermediate mesh to be generated. Figure 3 shows the five functions required to generate a new mesh in ALPS. These functions are described in more detail in Section 2.3. Note that the first four operate on the parallel octree, and only the last generates the mesh data structure.

When generating a mesh from an adapted octree, the interpolation of element fields between old and new meshes necessitates additional functions. The procedure for adapting the mesh works as follows. First, a given octree is coarsened and refined based on an application-dependent criterion, such as an error indicator. Next, the octree is "balanced" to enforce the 2-to-1 adjacency constraint. After these operations, a mesh is extracted so that the relevant finite element fields can be transferred between meshes. Following this, the adapted mesh is partitioned and the finite element fields are transferred to neighboring cores following their associated leaf partition. Figure 4 illustrates this process.

2.3 AMR FUNCTIONS

Below we highlight the key features of the functions used to build and adapt the octree and mesh in an application code. Here, "application code" refers to a code for the numerical discretization and solution of PDEs built on meshes generated from the octree.

NEWTREE. This algorithm is used to construct a new octree in parallel. This is done by having each core grow an octree to an initial coarse level. (This level is several units smaller than the level used later in the simulation.) At this point, each core has a copy of the coarse octree, which is then divided evenly between cores. The cores finish by pruning the parts of the tree they do not own, as determined by the Morton order. This is an inexpensive operation which requires no communication.

COARSENTREE/REFINETREE. Both COARSENTREE and REFINETREE work directly on the octree and are completely local operations requiring no communication. On each core, REFINETREE traverses the leaves of the local partition of the octree, querying the application code whether or not a given leaf should be refined. If so, eight new leaves are added to the level beneath the queried octant. COARSENTREE follows a similar approach examining the local partition of the octree for eight leaves from the same parent that the application

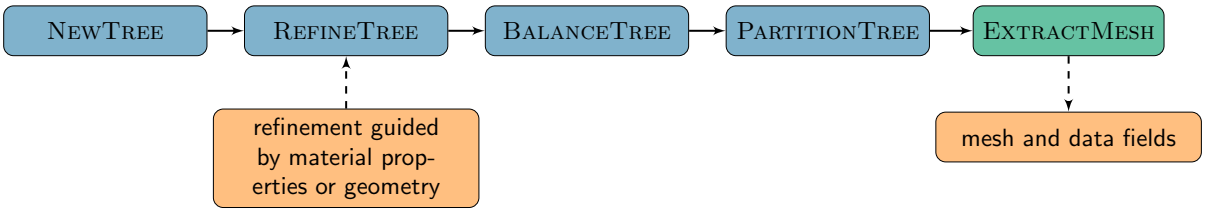

Figure 3: Functions for initial mesh generation. Blue boxes correspond to functions that operate on the octree only; cyan boxes denote functions that act between the octree and the mesh; and mesh and data field operations are enclosed in orange boxes. Solid arrows represent the flow of function calls; dashed arrows signify the input and output of mesh and/or data fields.

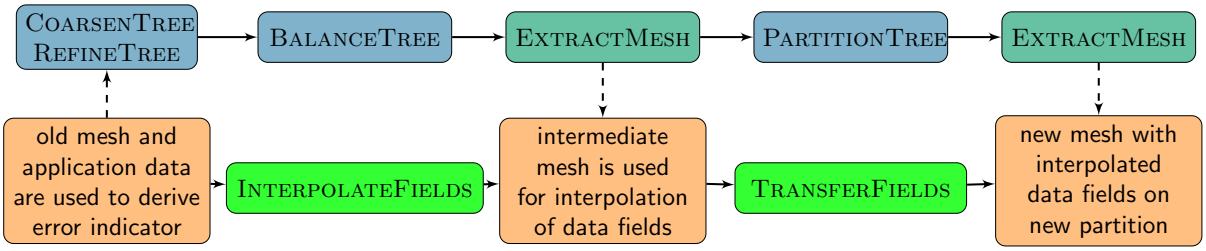

Figure 4: Functions for mesh adaptation. Colors and arrows as in Figure 3, augmented by green boxes for functions that act on the mesh and the application data fields only.

code has marked for coarsening. Note that we do not permit coarsening of a set of leaf octants that are distributed across cores. This is a minor restriction, since the number of such leaf sets is at most one less than the number of cores. Both CoarsenTree and RefineTree work recursively; that is, multiple levels of leaves can be removed or added in one invocation of the function.

BalanceTree. Enforcing the 2-to-1 size difference constraint between adjacent elements, also known as balancing the tree, is done with the parallel prioritized ripple propagation algorithm described in [20]. The algorithm uses a buffer to collect the communication requests as it balances the octree one refinement level at a time. This buffering aggregates all of the communication so that the number of communication rounds scales linearly with the number of refinement levels.

PartitionTree. Dynamic partitioning of the octree for load-balance is a key operation that has to be performed frequently throughout a simulation as the mesh is adapted. The goal of this function is to assign an equal number of elements to each core while keeping the number of shared mesh nodes between cores as small as possible. The space-filling curve offers a natural way of partitioning the octree, and hence mesh, among cores. The curve is divided into one segment per core according to the total ordering. The result is a partition with good locality properties, i.e., neighboring elements in the mesh tend to be found on the same core.

ExtractMesh. This function builds the mesh from a given octree and sets up the communication pattern for the application code. Unique global ordering of the elements and degrees of freedom of the mesh are determined and the relationship between the elements and nodes is established. Hanging nodes do not have unknowns associated with them, and therefore are not part of the global degrees of freedom. Their dependence on the global degrees of freedom, which is required to enforce the continuity of the finite element data fields, is also determined in this function. Ghost layer information (one layer of elements adjacent to local elements) from remote cores is also gathered.

InterpolateFields. This function is used to interpolate finite element data fields from one mesh to a new mesh that has been created by at most one level of coarsening and refinement. For simple interpolation between two trilinear finite element meshes, there is no global communication required to execute this step, given the value of ghost degrees of freedom. Once finished the cores gather the information for their ghost degrees of freedom by communicating with their neighboring cores.

TransferFields. This function is the operation on the data fields analogous to PartitionTree. Following the Morton ordering among the degrees of freedom, the data associated with element nodes is transferred between cores to complete the load-balancing stage. At the end of this process every core has obtained the data for all elements it owns and discarded what is no longer relevant due to the changed partition.

3 Dynamic Mesh Adaptation

We target PDE applications in which the locations of solution features evolve over time, necessitating dynamic mesh adaptivity. In this section we describe how we treat adaptivity for time-dependent problems and how we decide on which elements to mark for coarsening and refinement.

3.1 Marking Elements for Coarsening and Refinement

Most of the algorithms required for dynamic mesh coarsening and refinement have been discussed in Section 2.3. However, given an error indicator η_e for each element e, we still need a method to mark elements for coarsening and refinement. Since we wish to avoid a global sort of all elements according to their error indicators, we propose a strategy that adjusts global coarsening and refinement thresholds through collective communication. This process iterates until the overall number of elements expected after adaptation lies within a prescribed tolerance around a target. The main objective of the algorithm MarkElements is to find thresholds $\eta_{\text{coarsen}} < \eta_{\text{refine}}$ such that:

- Every element e with $\eta_e < \eta_{\text{coarsen}}$ is coarsened.
- Every element e with $\eta_e > \eta_{\text{refine}}$ is refined.
- Obtain $\approx N$ total elements after adaptation.

MarkElements. Assume a given error indicator η_e for each element e and a target number N of elements.

1. Select a coarsening threshold η_{coarsen} based on the mean value $\bar{\eta}$ and standard deviation σ of η_e.
2. Choose $\gamma < 1$ and prepare a binary search for the refinement threshold by setting $k := 0$, $\eta_{\text{low}} := \eta_{\min}$, $\eta^{(0)} := \eta_{\text{high}} := \eta_{\max}$.
3. Count the overall expected number of elements $N^{(k)}$ determined by η_{coarsen} and $\eta_{\text{refine}} := \eta^{(k)}$. This requires one collective communication call.
4. If $N^{(k)} < \gamma N$ then set $\eta_{\text{high}} := \eta^{(k)}$, $k := k + 1$, $\eta^{(k)} := \frac{1}{2}(\eta_{\text{low}} + \eta_{\text{high}})$ and go to Step 3.
5. If $N^{(k)} > N$ then set $\eta_{\text{low}} := \eta^{(k)}$, $k := k + 1$, $\eta^{(k)} := \frac{1}{2}(\eta_{\text{low}} + \eta_{\text{high}})$ and go to Step 3.

In principle the coarsening threshold could be determined by a binary search as well. We found the simplified approach based on statistical properties of the error indicator to be sufficient.

To determine the number of expected elements in Step 3, we compute the global sum of the local element counts. We limit the number of iterations for the search loop to 20, which is seldom attained in practice.

The factor γ in Step 4 should be chosen smaller than 1.0 (we use 0.97 in our implementation) for two reasons:

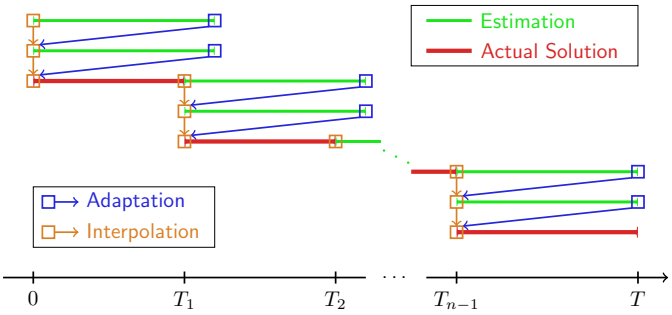

Figure 5: Outline of window-based mesh adaption scheme. The total time T of the simulation is covered by n time windows, which overlap by a prescribed amount (10% of the window length in this paper). The PDEs are integrated in time over a window m times for error estimation purposes, and finally once for archival purposes on the final adapted mesh. At the end of each error estimation integration, the mesh is adapted according to error indicators accumulated over the integration through the time window. The mesh is repartitioned, and the initial conditions for the window are regenerated by interpolation from the old to the new mesh. Then, integration over the window is repeated, with a possibly different time step size. The figure illustrates the case $m = 2$; in the examples in this paper, we use $m = 1$. In practice the windows have different lengths due to the fixed number of time steps used for the first integration.

The number of expected elements changes with η_{refine} in discrete steps only, and we take into account that the subsequent enforcement of the 2-to-1 balance condition may increase slightly the number of elements.

We choose the target N based on the memory available to every core; this way, the number of elements is kept approximately constant during the simulation (in practice the variations are within just 2%).

3.2 Window-Based Dynamic Mesh Adaptation

This section briefly describes how we obtain error indicators for time-dependent computations in which the mesh changes dynamically. Other approaches for dynamic mesh adaptation than the one given here are certainly possible. Our approach is based on ideas discussed in [18]. The mesh is adapted at the transition between successive intervals called *time windows*. The minimum length of a time window is one time step. However, adapting every time step is usually wasteful. Therefore, we allocate a fixed number of time steps to each window (in this paper, 32). The size of the time step varies, and thus the lengths of the windows vary. We repeat the numerical integration of the PDEs over a time window one or more times to accumulate error estimates, and we adapt the mesh based on this information. Only the last integration over the window is retained to represent the actual numerical solution of the PDEs, as illustrated in Figure 5. The benefit of this approach is that the "archival" integration over each time window is conducted on a mesh tailored to the evolving solution.

4 EXAMPLE: ADVECTION DIFFUSION EQUATION

In this section we study the performance of ALPS on parallel dynamic mesh adaptation for a time-dependent advection-diffusion equation. Advection-diffusion equations are used to model a wide range of transport phenomena in natural and engineered systems. After a brief description of the initial-boundary value problem and its discretization, we study the volume of adaptivity, weak and strong scalability, and the overall performance of the method. All runs (besides the profiling run shown in Figure 12) are conducted on Ranger, the 0.5 Petaflop/s, 123 Terabyte, 62,976-core Sun/AMD system at the Texas Advanced Computing Center (TACC).

4.1 PROBLEM SETUP

To study the performance of ALPS, we consider the transient advection-diffusion equation

$$\frac{\partial C}{\partial t} + \mathbf{u} \cdot \nabla C - \nabla \cdot \kappa \nabla C = 0, \quad \mathbf{x} \in \Omega,\ t \in (0,T) \qquad (1)$$

with given initial condition $C(\mathbf{x},0) = C_0(\mathbf{x})$ and appropriate boundary conditions. The PDE (1) models a time-dependent unknown $C(\mathbf{x},t)$ (e.g., temperature or concentration) in a 3D-domain Ω. The unknown is transported by a given flow field $\mathbf{u}$ and at the same time diffuses with rate $\kappa \geq 0$.

For all of the numerical results in this section, we solve (1) in $\Omega = [0,30] \times [0,30] \times [0,30]$ with $\kappa = 0.001$, $T = 2$, $\mathbf{u} = [1\ 1\ 0]^T$ and $C_0(\mathbf{x}) = 1/2\left(1 - \tanh\left(\alpha\left(\|\mathbf{x} - \mathbf{x}_0\|^2 - 5\right)\right)\right)$ where $\mathbf{x}_0 = [10\ 10\ 10]^T$ and $\alpha = 0.1$. Homogeneous Dirichlet conditions are taken on all boundaries.

The advection-diffusion equation (1) represents a challenging test problem for parallel mesh adaptivity when convection dominates, since the equation acquires hyperbolic character (which transports sharp fronts). In this case, the unknown field C can be highly localized in space, and thus very high local mesh resolution may be required to avoid numerical (and thus unphysical) dissipation. On the other hand, a very coarse mesh may be sufficient in other regions. As time evolves, the mesh will have to adapt to the changing distribution of C. Namely, the mesh can be coarsened away from high gradient regions, and it must be refined near those regions. After each adaptation step, the computational load and memory requirements on each core can change drastically. To balance the load and avoid exceeding local memory, the mesh must be redistributed uniformly among all cores. A sequence of meshes that are adapted to the unknown C is shown in Figure 6, illustrating the substantial change in the mesh over time.

4.2 DISCRETIZATION AND STABILIZATION

We discretize (1) in space with trilinear finite elements on octree-based hexahedral meshes. As mention previously, these meshes are non-conforming, i.e., they contain hanging nodes on element faces and edges. To ensure continuity of C, these nodes do not correspond to degrees of freedom. Rather, the value of hanging node variables is uniquely determined by the values at the neighboring independent nodes. This leads to algebraic constraints that are used to eliminate the hanging nodes at the element level. It is well known that Galerkin finite element discretizations of (1) require stabilization to suppress spurious oscillations in advection-dominated flows (i.e., when κ is very small compared to $\mathbf{u}$). Therefore we use the streamline upwind/Petrov-Galerkin (SUPG) method [6]. Since we are mainly interested in the advection-driven regime of (1), we use an explicit predictor-corrector time integration [13].

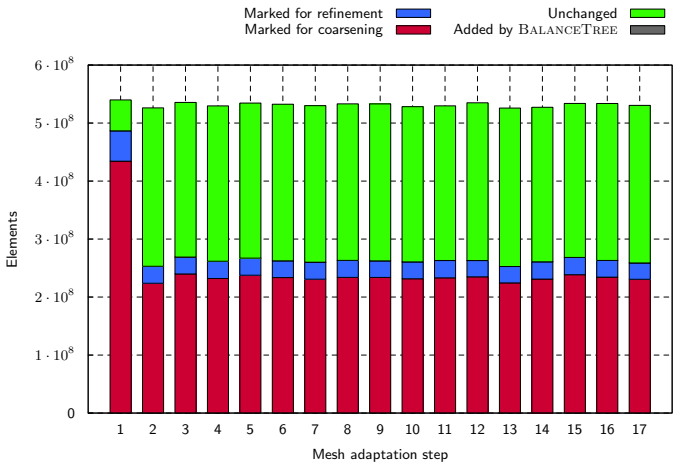

Figure 7: The number of elements that are refined (blue), coarsened (red), created to respect the 2:1 balance condition (brown, barely visible), and unchanged (green), after adaptation of each time window. Simulation ran on 4096 cores with approximately 131,000 elements per core.

4.3 EXTENT OF MESH ADAPTIVITY

Figure 7 shows the number of elements that are coarsened, refined, and stay the same in each mesh adaptation step (i.e., per time window), for a simulation on 4096 cores. The number of adapted elements amount to about half of the total number in the mesh at every adaptation step. Figure 8 depicts the distribution of elements among the refinement levels for selected time windows. The final meshes contain elements on 10 levels of the octree, leading to a factor of 512 variation in element edge length. These figures illustrate the significant extent of adaptivity over the course of a simulation. This is worth keeping in mind when examining the scalability results in subsequent sections.

4.4 SCALABILITY

In our analysis below, we study both *isogranular* (often called *weak*) scalability and *fixed-size* (or *strong*) scalability. As the name suggests, for fixed-size scalability a fixed

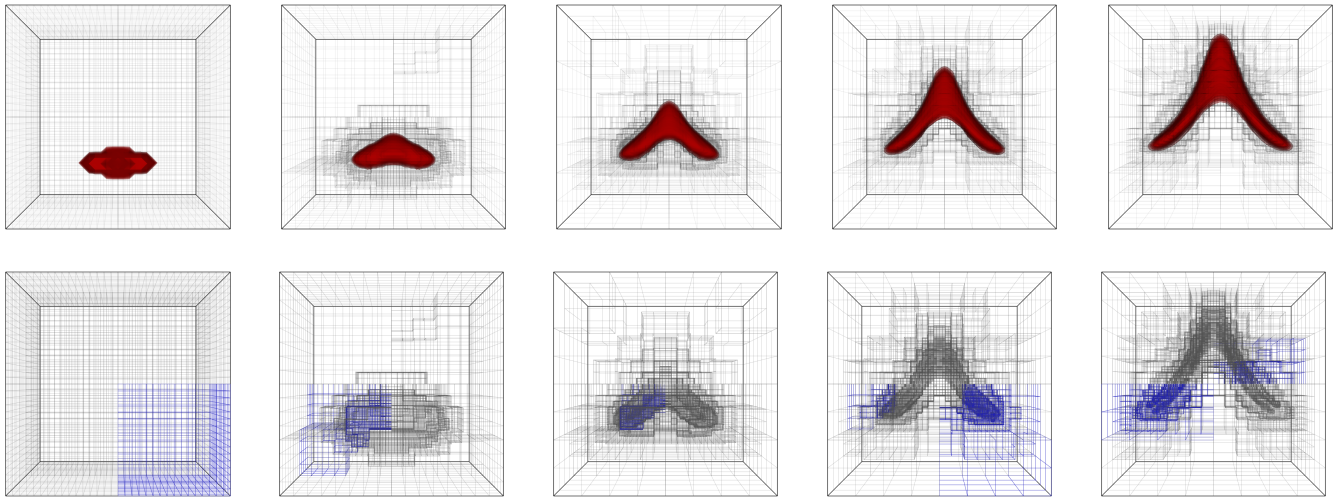

Figure 6: Illustration of mesh adaptivity for advection-diffusion equation (1) on 8 cores with a vertical flow field. The figure shows snapshots of isosurfaces of the transported quantity C at five time instants (upper row) and corresponding adapted meshes (lower row). The blue elements in the lower row belong to Core #3, indicating the dynamic evolution of the mesh partition.

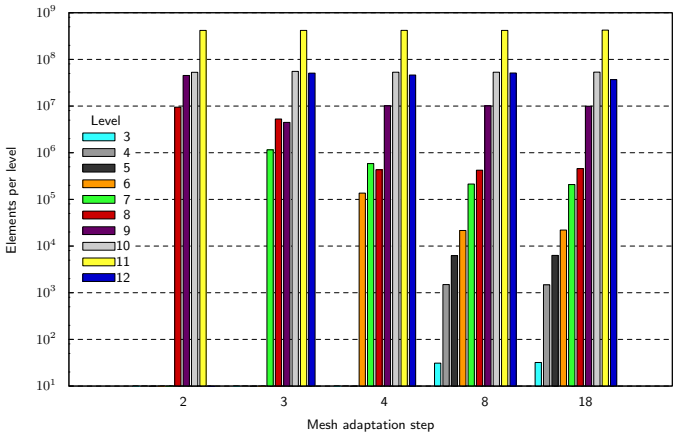

Figure 8: The number of elements within each level of the octree for selected time windows, for a simulation on 4096 cores with approximately 131,000 elements per core. The octree is 10 levels deep for late-time meshes, which corresponds to a variation of element edge length by a factor of 512 across the same mesh.

problem is solved on an increasing number of cores and the speedup in runtime is reported. Isogranular scalability increases the problem size and the number of cores simultaneously so that problem size per core remains the same. Both notions are valuable performance indicators for parallel algorithms. For fixed-size scaling studies, as the load per core becomes too small, the overall runtime is dominated by the communication cost. Thus, we give speedup curves for only a certain range of core numbers, and then provide several curves corresponding to several different problem sizes. The total end-to-end run time is always used to calculate speedup, including the initial octree setup, mesh generation, and all phases of adaptivity.

Scaling up the problem size N for isogranular scalability studies is straightforward for problems involving discretized PDEs, since one needs only to refine the mesh. However, it exposes a lack of *algorithmic scalability* in numerical and graph algorithms whose work or communication increase superlinearly in N. This applies to algorithms for dynamic meshing and load balancing that require the processing of global information by individual cores, as well as to numerical algorithms such as Krylov or fixed-point equation solvers for which the number of iterations grows with problem size.

For dynamic mesh adaptivity, it is usually difficult to keep the number of elements per core constant as the problem size increases. Our algorithm MARKELEMENTS performs well in this respect. Still, to provide a strict test of isogranular scalability, we report the total work performed for any given problem size, where *total work* is defined as the total number of elements integrated over all cores and all time steps. This measure normalizes for any adaptivity-driven imbalances in the number of time steps or elements per core as we scale up weakly. Thus "total work" becomes synonymous with problem size, and we can assess the isogranular scalability of our implementa-

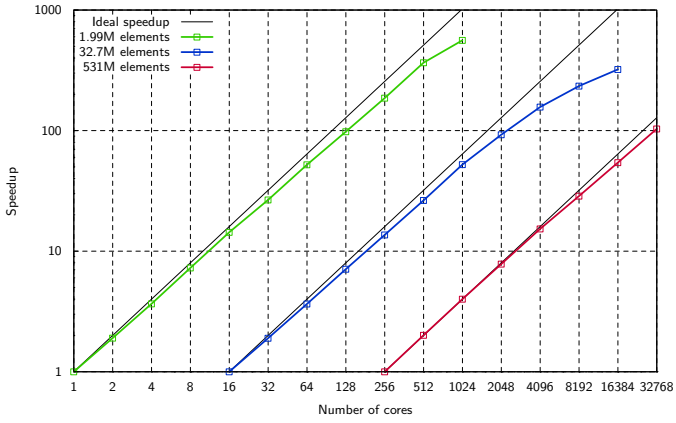

Figure 9: Fixed-size scalability: Speedup based on total runtime plotted against the number of cores for three different problem sizes.

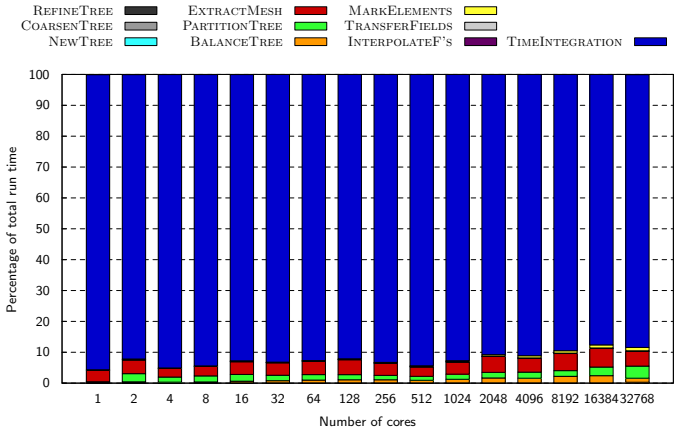

Figure 10: Isogranular scalability: Breakdown of total run time into different components related to numerical PDE integration (blue) and AMR functions (all other colors), with increasing number of cores from 1 to 32,768. Problem size increases isogranularly at roughly 131,000 elements per core (largest problem has approximately 4.29 billion elements). The overall runtime is dominated by the numerical integration. The most expensive operation related to AMR is ExtractMesh, which uses up to 6% of the runtime. Overall, AMR consumes about 10% or less of the overall time.

tion by judging whether the total work divided by the total run time and the number of cores remains constant as the number of cores is increased.

4.4.1 Fixed-Size Scaling

Figure 9 shows the speedups for small, medium-size, and large test problems as the number of cores increases. The initial meshes on 1, 16, and 256 cores contain approximately 2.1 million elements per core. This number is kept approximately constant throughout the whole run using the threshold adjustment algorithm MarkElements described in Section 3.1.

We selected these problem sizes based on the multicore architecture of Ranger. Every compute node has 32 GB of main memory and 16 cores. The size of each problem is chosen such that it uses 1 core per node for the initial number of cores (1, 16, 256). This provides optimal conditions for these initial runs since every core has exclusive access to processor memory and the node's network interface. The first four scaling steps keep the number of blades constant and increase the number of cores used per node to 16, at the same time increasing the sharing of resources. Once all 16 cores are used on a node, we increase the number of nodes in powers of 2.

The speedup is nearly optimal over a wide range of core counts. For instance, solving the small test problem (green line) on 512 cores is still 366 times faster than the solution on a single core. Similarly, the overall time for solving the medium-size problem (blue line) on 1024 cores results in a speedup of more than 52 over the runtime for 16 cores (optimal speedup is 64).

4.4.2 Isogranular Scaling

Figures 10 and 11 provide insight into the isogranular scalability of the advection-diffusion test problem. Figure 10 depicts the breakdown of the overall runtime into the time consumed by mesh initialization, the time needed by the various functions to adapt the mesh, and the time spent in numerical time integration, for weak scaling from 1 to 32,768 cores. The results indicate that *all mesh adaptivity functions*—including refinement, coarsening, interpolation, field transfer, rebalancing, repartitioning, and mesh extraction—impose little overhead on the PDE solver. Only for 16K and 32K cores does the total cost of AMR exceed 10% of end-to-end run time, and even then just barely. In fact, our scalar, linear, low-order-discretized, explicitly-time-advanced advection-diffusion test problem (1) imposes the most strenuous test of AMR, since it offers little numerical work to amortize mesh adaptivity. Moving to a vector-valued or nonlinear PDE, a high-order discretization, or an implicit solver will significantly increase the numerical work, suggesting that the AMR overhead will be negligible.

As illustrated in the lower row of Figure 6, PartitionTree completely redistributes the octree (and thus the mesh) among all cores (we do not impose an explicit penalty on data movement in the underlying partitioning algorithm). In general, the entire octree is reshuffled among the cores after adaptation, which amounts to large amounts of sent and received data using a one-to-one communication pattern. Interestingly, Figure 10 demonstrates that the time for PartitionTree remains essentially constant for all core counts beyond 1 (for which partitioning is unnecessary). This is also reflected in the timings for TransferFields, which do not show up in Figure 10 even though complete finite element data fields are redistributed through the network.

Figure 11 displays the parallel efficiency for isogranular scaling from 1 to 32,768 cores. Here, parallel efficiency is defined as the total work per number of cores per total

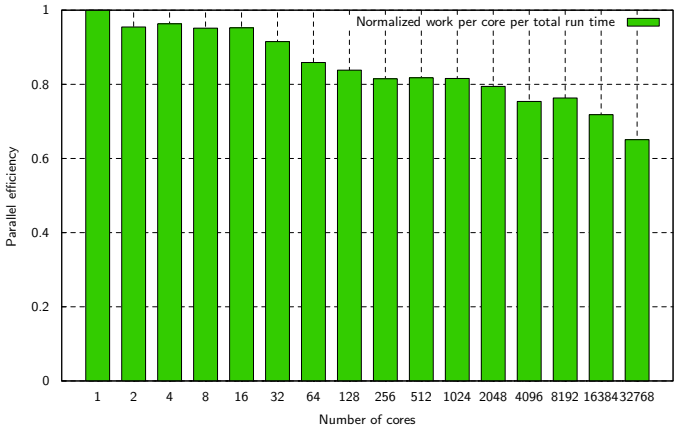

Figure 11: Isogranular scalability: Parallel efficiency measured in total work (i.e. total processed elements) per core per total run time, with increasing number of cores from 1 to 32,768 (problem size approximately 131,000 elements per core). Despite a 32-thousand-fold increase in problem size and number of cores, parallel efficiency remains above 65%.

run time for a given core count, normalized by the total work per total run time for a single core. A parallel efficiency of 1.0 indicates perfect isogranular scalability. For example, if the adaptivity process resulted in exactly twice as many elements created and the same number of time steps when doubling the core count and reducing the acceptable error (to increase problem size), then we would simply use the run time as a measure of parallel efficiency. Otherwise, normalizing by the number of elements per core and the number of time steps accommodates any possible vagaries of the adaptivity process. As can be seen in the figure, despite a 32-thousand-fold increase in problem size and number of cores, parallel efficiency remains above 65%. Again, we must keep in mind the strenuous nature of this (scalar, linear, low-order-discretized, explicitly-integrated) test problem.

4.5 Load Balance

We use the TAU utility [17] to assess load balance. Figure 12 shows the decomposition of the runtime on each of 64 cores into major functions of the code. The dominant function—element matrix-vector products—has nearly optimal load balance across the 64 cores, as shown by the close alignment of the blue bars. Time spent in other, less dominant, functions shows good load balance as well. The excellent scalings presented in Section 4.4 reflect this good load balance.

5 Conclusions

Because of the complex data structures and large volumes of communication required, the scalability of dynamic AMR to tens of thousands of processors has long been considered a challenge. We have presented ALPS, a library for dynamic mesh adaptation and redistribution that uses parallel octree-based hexahedral finite element meshes and dynamic load balancing based on space-filling curves. We have presented results from the solution of advection-dominated transport problems on TACC's Ranger system that demonstrate excellent load balance and scalability on up to 32K cores and 4 billion elements. The results indicate that all overheads due to mesh adaptivity—including refinement, coarsening, rebalancing, redistribution, and repartitioning—consume 10% or less of the overall runtime. We anticipate that more complex and/or nonlinear PDEs that will require higher order and implicit discretizations will render the AMR overhead invisible, and permit ALPS to scale to hundreds of thousands of cores.

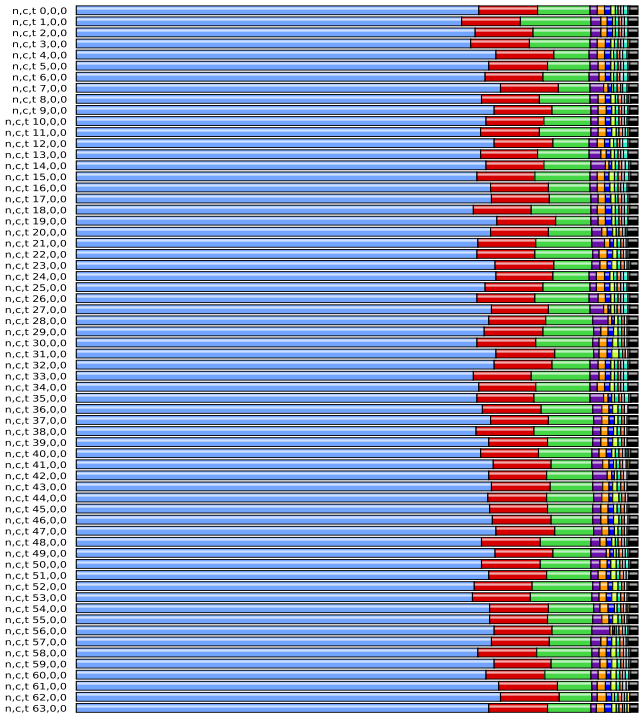

Figure 12: Performance profile of a complete run of ALPS on 64 cores of TACC's Lonestar (Intel Woodcrest) system. Each row shows time spent in different functions; for example, blue bars correspond to element-level matrix multiplication within the time integration; red bars correspond to the error estimation; green bars correspond to `MPI_Waitall` calls.

Acknowledgements

This work was partially supported by NSF (grants OCI-0749334, DMS-0724746, CNS-0619838, CCF-0427985), DOE SC's SciDAC program (grant DE-FC02-06ER25782), DOE NNSA's PSAAP program (cooperative agreement DE-FC52-08NA28615), and AFOSR's Computational Math program (grant FA9550-07-1-0480). We acknowledge many helpful discussions with George Biros and thank TACC for their outstanding support, in particular Bill Barth, Karl Schulz, and Romy Schneider.

REFERENCES

[1] M. AINSWORTH AND J. T. ODEN, *A posteriori error estimation in finite element analysis*, Pure and Applied Mathematics, John Wiley & Sons, New York, 2000.

[2] I. BABUŠKA AND T. STROUBOULIS, *The finite element method and its reliability*, Numerical Mathematics and Scientific Computation, The Clarendon Press, Oxford University Press, New York, 2001.

[3] D. BAILEY, *The 1997 Petaflops Algorithms Workshop*, Computational Science & Engineering, IEEE, 4 (Apr-Jun 1997), pp. 82–85.

[4] 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.

[5] R. BECKER AND R. RANNACHER, *An optimal control approach to a posteriori error estimation in finite element methods*, Acta Numerica, 10 (2001), pp. 1–102.

[6] A. N. BROOKS AND T. J. R. HUGHES, *Streamline upwind Petrov-Galerkin formulations for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equations*, Computer Methods in Applied Mechanics and Engineering, 32 (1982), pp. 199–259.

[7] A. CALDER, B. CURTIS, L. DURSI, B. FRYXELL, G. HENRY, P. MACNEICE, K. OLSON, P. RICKER, R. ROSNER, F. TIMMES, H. TUFO, J. TRURAN, AND M. ZINGALE, *High-performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors*, in Supercomputing, ACM/IEEE 2000 Conference, 2000, pp. 56–56.

[8] P. COLELLA, J. BELL, N. KEEN, T. LIGOCKI, M. LIJEWSKI, AND B. VAN STRAALEN, *Performance and scaling of locally-structured grid methods for partial differential equations*, Journal of Physics: Conference Series, 78 (2007), pp. 1–13.

[9] P. COLELLA, T. H. DUNNING, W. D. GROPP, AND D. E. KEYES, *A Science-Based Case for Large-Scale Simulation, Volume 2*. Office of Science, U.S. Department of Energy, September 2004.

[10] L. DEMKOWICZ, J. KURTZ, D. PARDO, M. PASZYŃSKI, W. RACHOWICZ, AND A. ZDUNEK, *Computing with hp Finite Elements II. Frontiers: Three-Dimensional Elliptic and Maxwell Problems with Applications*, CRC Press, Taylor and Francis, 2007.

[11] L. F. DIACHIN, R. HORNUNG, P. PLASSMANN, AND A. WISSINK, *Parallel adaptive mesh refinement*, in Parallel Processing for Scientific Computing, M. A. Heroux, P. Raghavan, and H. D. Simon, eds., SIAM, 2006, ch. 8.

[12] J. E. FLAHERTY, R. M. LOY, M. S. SHEPHARD, B. K. SZYMANSKI, J. D. TERESCO, AND L. H. ZIANTZ, *Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws*, Journal of Parallel and Distributed Computing, 47 (1997), pp. 139–152.

[13] T. J. R. HUGHES, *The Finite Element Method*, Dover, New York, 2000.

[14] B. KIRK, J. W. PETERSON, R. H. STOGNER, AND G. F. CAREY, *libMesh: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations*, Engineering with Computers, 22 (2006), pp. 237–254.

[15] A. LASZLOFFY, J. LONG, AND A. K. PATRA, *Simple data management, scheduling and solution strategies for managing the irregularities in parallel adaptive hp finite element simulations*, Parallel Computing, 26 (2000), pp. 1765–1788.

[16] C. D. NORTON, G. LYZENGA, J. PARKER, AND R. E. TISDALE, *Developing parallel GeoFEST(P) using the PYRAMID AMR library*, tech. rep., NASA Jet Propulsion Laboratory, 2004.

[17] S. SHENDE AND A. D. MALONY, *The TAU parallel performance system*, International Journal of High Performance Computing Applications, 20 (2006), pp. 287–331.

[18] S. SUN AND M. F. WHEELER, *Mesh adaptation strategies for discontinuous galerkin methods applied to reactive transport problems*, in Proceedings of the International Conference on Computing, Communication and Control Technologies, H.-W. Chu, M. Savoie, and B. Sanchez, eds., 2004, pp. 223–228.

[19] 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, (2007). To appear.

[20] T. TU, D. R. O'HALLARON, AND O. GHATTAS, *Scalable parallel octree meshing for terascale applications*, in Proceedings of SC2005, 2005.

[21] A. M. WISSINK, D. A. HYSOM, AND R. D. HORNUNG, *Enhancing scalability of parallel structured AMR calculations*, in Proceedings of the International Conference on Supercomputing 2003 (ICS'03), San Francisco, CA, June 2003, pp. 336–347.