Dendro: Parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees

Rahul S. Sampath*, Santi S. Adavani†, Hari Sundar†, Ilya Lashuk*, and George Biros*
* Georgia Institute of Technology, Atlanta, GA 30332
† University of Pennsylvania, Philadelphia, PA 19104
Rahul.Sampath@gatech.edu, adavani@seas.upenn.edu, hsundar@seas.upenn.edu,
ilashuk@cc.gatech.edu, gbiros@cc.gatech.edu

***Abstract*—In this article, we present Dendro, a suite of parallel algorithms for the discretization and solution of partial differential equations (PDEs) involving second-order elliptic operators. Dendro uses trilinear finite element discretizations constructed using octrees. Dendro, comprises four main modules: a bottom-up octree generation and 2:1 balancing module, a meshing module, a geometric multiplicative multigrid module, and a module for adaptive mesh refinement (AMR). Here, we focus on the multigrid and AMR modules. The key features of Dendro are coarsening/refinement, inter-octree transfers of scalar and vector fields, and parallel partition of multilevel octree forests. We describe a bottom-up algorithm for constructing the coarser multigrid levels. The input is an arbitrary 2:1 balanced octree-based mesh, representing the fine level mesh. The output is a set of octrees and meshes that are used in the multigrid sweeps. Also, we describe matrix-free implementations for the discretized PDE operators and the intergrid transfer operations. We present results on up to 4096 CPUs on the Cray XT3 ("BigBen") , the Intel 64 system ("Abe"), and the Sun Constellation Linux cluster ("Ranger").**

I. Introduction

Second-order elliptic operators, like the Laplacian operator, are ubiquitous in computational science and engineering. For example, they model electromagnetic interactions, diffusive transport, viscous dissipation in fluids, and stress-strain relationships in solid mechanics [13], [21]. Meeting the need for high-resolution simulation of such PDEs requires scalable non-uniform discretization methods and scalable solvers for the finite-dimensional linear operators that result upon discretization of the elliptic operators. Unstructured finite element meshes are commonly used for non-uniform discretizations but several challenges remain with respect to cache-access efficiency, parallel meshing, partitioning, and coarsening [3], [19], [40]. Octree-based finite element meshes strike a balance between the simplicity of structured meshes and the adaptivity of unstructured meshes [2], [6], [30], [34]. On multithousand-CPU platforms, the resulting octree-based discretized operators must be "inverted" using iterative solvers. Multigrid algorithms (geometric and algebraic) are iterative methods that provide a powerful mathematical framework that allows the construction of solvers with optimal algorithmic complexity [37]: their convergence rate is independent of the mesh size. Numerous sequential and parallel implementations for both geometric and algebraic multigrid methods on structured and unstructured meshes are available [1], [9], [10], [15], [16], [20], [22], [24], [26]. In this article, we focus on implementing a parallel geometric multiplicative multigrid scheme on octree meshes.

Related work on parallel multigrid for non-uniform meshes: Excellent surveys on parallel algorithms for multigrid can be found in [12] and [23].[1] Here, we give a brief (and incomplete!) overview of distributed memory message-passing based parallel algebraic and geometric multigrid algorithms. The advantages of algebraic multigrid is that it can be used in a black-box fashion with unstructured grids and operators with discontinuous coefficients. The disadvantage of existing algebraic multigrid implementations is the relatively high (compared to the solution time) setup costs. The main advantage of geometric multigrid is that it is easier to devise coarsening and intergrid transfers with low overhead. Its disadvantage is that it is harder to use in a black-box fashion.

We are interested in developing a parallel multigrid method for highly non-uniform discretizations of elliptic PDEs. Currently, the most scalable methods are based on graph-based methods for coarsening, namely maximal-matchings. Examples of scalable codes that use such graph-based coarsening include [1] and [26]. Another powerful code for algebraic multigrid is BoomerAMG, which is a part of the Hypre package [15], [16]. BoomerAMG has been used extensively to solve problems with a variety of operators on structured and unstructured grids. However, the associated constants for constructing the mesh and performing the calculations can be quite large. The high-costs related to partitioning, setup, and accessing generic unstructured grids, has motivated us to design geometric multigrid for octree-based data structures.

Parallel geometric multigrid algorithms for non-uniform discretizations have been proposed in the past. A modestly parallel (using up to 16 processors) additive multigrid solver for three dimensional Poisson problems on adaptive hierarchical Cartesian grids was presented in [10], [27] and the details of the sequential version are presented in [20], [28]. In that approach, the adaptive grids are constructed by recursively splitting each cell/element into three subcells in each coordinate direction unlike octrees/quadtrees, which are constructed by recursively splitting each cell/element into 2 subcells in each coordinate direction. Hence, the number of grid points grows like 3^d instead of 2^d in d-dimensions for each level of refinement. In [5], the authors describe a Poisson solver with excellent scalability on up to 1024 processors. Strictly speaking, it is not a bona fide

[1] We do not attempt to review the extensive literature on methods for adaptive mesh refinement. Our work is restricted to construction of efficient intergrid transfers between different octree meshes and not on error estimation.

SC2008 November 2008, Austin, Texas, USA 978-1-4244-2835-9/08 $25.00 ©2008 IEEE

multigrid solver as there is no iteration between the multiple levels. Instead, it is based on local independent solves in nested grids followed by global corrections to restore smoothness. It is similar to block structured grids and its extension to arbitrarily graded grids is not immediately obvious. One of the largest calculations using finite element meshes was reported in [7], using conforming discretizations and geometric multigrid solvers on semi-structured meshes. That approach is highly scalable for nearly structured meshes. However, the scheme does not support arbitrarily non-uniform meshes.

Multigrid methods on octrees have been proposed and developed for sequential and modestly parallel adaptive finite element implementations [6], [18]. A characteristic of octree meshes is that they contain "hanging" vertices.[2] In [34], we presented a strategy to tackle these hanging vertices and build conforming, trilinear finite element discretizations on these meshes. That algorithm scaled up to four billion octants on 4096 CPUs on a Cray XT3 at the Pittsburgh Supercomputing Center. We also showed that the cost of applying the Laplacian using this framework is comparable to that of applying it using a direct indexing regular grid discretization with the same number of elements. ***To our knowledge, there is no work on large scale, parallel, octree-based, matrix-free, geometric multiplicative multigrid solvers for finite element discretizations that has scaled to thousands of processors. Here, we build on our previous work and present such a parallel geometric multigrid scheme.***

Terminology: In the following sections, we will use the term "*MatVec*" to denote a matrix-vector multiplication, we will use "*octants*" to refer to both octree nodes and the corresponding finite elements and "*vertices*" to refer to element vertices. We will use "*CPUs*" to refer to message passing processes. We use the phrase "bottom-up approach" to refer to an algorithm that uses the finest octree as its input. In contrast, we use the phrase "top-down approach" to refer to an algorithm that starts with the coarsest octree and proceeds to the finest octree. We often refer to "*sorted*" octrees, these are lists of octants that are sorted according to the Morton numbers (explained later) of the octants.

Contributions: Our goal is to minimize storage requirements, obtain low setup costs, and achieve end-to-end[3] parallel scalability:

- We propose a parallel global **coarsening algorithm** to construct a hierarchy of nested 2:1 balanced octrees and their corresponding meshes starting with an arbitrary 2:1 balanced fine octree. We do not impose any restrictions on the number of meshes in this sequence or the size of the coarsest mesh. The process of constructing coarser meshes from a fine mesh is harder than iterative global refinements of a coarse mesh for the following reasons: **(1)** We must ensure that the coarser grids satisfy the 2:1 balanced constraint as well; while this is automatically satisfied for the case of global refinements, the same does not hold for global coarsening. **(2)** Even if the input to the coarsening function is load balanced, the output may not be so. Hence, global coarsening poses additional challenges with partitioning and load balancing. However, this bottom-up approach is more natural for typical PDE applications in which only some discrete representation (e.g., material properties defined at certain points) is available.
- Transferring information between successive multigrid levels is a challenging task on unstructured meshes and especially so in parallel since the coarse and fine grids need not be aligned and near neighbor searches are required. Here, we describe a matrix-free algorithm for **intergrid transfer operators** that uses the special properties of an octree data structure to circumvent these difficulties.
- We have integrated the above mentioned components in a parallel matrix-free implementation of a geometric multiplicative multigrid method for finite elements on octree meshes. Our MPI-based implementation, `Dendro` has scaled to billions of elements on thousands of CPUs even for problems with large contrasts in the material properties. `Dendro` ***is an open source code that can be downloaded from*** [33]. `Dendro` is tightly integrated with PETSc [4].

Limitations of `Dendro`*:* Here, we summarize the limitations of the proposed methodology. **(1)** The restriction and prolongation operators and the coarse grid approximations are not robust for problems with discontinuous coefficients. **(2)** The method does not work for strongly indefinite elliptic operators (e.g., high-frequency Helmholtz problems). **(3)** `Dendro` is limited to second-order accurate discretization of elliptic operators on the unit cube. Cubes/Cuboids of other dimensions can be handled easily using appropriate scalings but problems with complex geometries are not directly supported. **(4)** Only the intergrid transfers have been implemented in the AMR module (we have not implemented an error estimation algorithm). **(5)** Load balancing is not addressed fully (i.e., we have observed suboptimal scalings).

Organization of the paper: In Section II, we give a brief introduction to our octree data structure and our framework for handling "hanging" vertices; more details can be found in [34], [35]. In Section III, we describe the implementation of the multigrid method. In particular, we describe the coarsening algorithm and the implementation of the MatVecs for the restriction/prolongation multigrid operations. The intergrid transfer of the solution vector that is necessary for the AMR algorithm is presented in Section IV. In Section V, we present the results from fixed-size and iso-granular scalability experiments. Also, we compare our implementation with "BoomerAMG" [22], an algebraic multigrid implementation available in Hypre. In Section VI, we present the conclusions from this study and discuss ongoing and future work.

II. Background

Each octree node has a maximum of eight children. An octant with no children is called a "*leaf*" and an octant with one or more children is called an "*interior octant*". "*Complete*"

[2]Vertices of octants that coincide with the centers of faces or mid-points of edges of other octants are referred to as "hanging" vertices.

[3]By end-to-end, we collectively refer to the construction of octree-based meshes for all multigrid levels, restriction/prolongation, smoothing, coarse solve, and CG drivers.

octrees are trees in which every interior octant has exactly eight children. The only octant with no parent is the "*root*" and all other octants have exactly one parent. Octants that have the same parent are called "*siblings*". The depth of the octant from the root is referred to as its "*level*".[4] We use a "*linear*" octree representation (i.e., we exclude interior octants) using the Morton encoding scheme [11], [35]. An octant's configuration with respect to its parent is specified by its "*child number*": The octant's position relative to its siblings in a sorted list. This is a useful property that will be used frequently in the remaining sections. Any octant in the domain can be uniquely identified by specifying one of its vertices, also known as its "*anchor*", and its level in the tree. By convention, the anchor of an octant is its front lower left corner (a_0 in Figure 1). In order to perform finite element calculations on octrees, we impose a restriction on the relative sizes of adjacent octants. This is also known as the "*2:1 balance constraint*" (not to be confused with load balancing): No octant should be more than twice the size of any other octant that shares a corner, edge, or face with this octant. This constraint has been used in many other works as well [6], [8], [14], [17], [18], [24], [38], [39]. In [35], we presented a parallel algorithm for linear octree construction and 2:1 balancing, which we use in `Dendro`. Vertices that exist at the center of a face of an octant are called *face-hanging vertices*; vertices that are located at the mid-point of an edge are called *edge-hanging vertices*. The 2:1 balance constraint ensures that there is at most one hanging vertex on any edge or face. *In this work, we only deal with sorted, complete, linear, 2:1 balanced octrees*.

A pre-processing step for finite element computation is "*meshing*". *The construction of the element-to-vertex mappings, the construction of ghost and local vertex lists, and the construction of the scatter/gather operators for the near-neighbor communications that are required during the MatVec are all components of parallel meshing*. Our meshing algorithm is described in [34]. The key features of our meshing algorithm are that **(1)** we use a hanging-vertex management scheme (Figure 1) that allows MatVecs in a single tree traversal as opposed to multiple tree traversal required by the scheme in [38], and **(2)** we reduce the memory overhead by storing the octree in a compressed form that requires only one byte per octant (used to store the level of the octant). The element-to-mesh vertex mappings can be compressed at a modest expense of uncompressing this on the fly while looping over the elements to perform the finite element MatVecs. In our hanging-vertex management scheme[5], we do not store hanging vertices explicitly since they do not represent independent degrees of freedom in a FEM solution. If the i-th vertex of an element is hanging, then the index corresponding to this vertex is mapped to the i-th vertex of the parent[6] of this element instead. Thus, if a hanging vertex is shared between 2 or more elements, then in each element it might point to a different index. Using this scheme, we construct

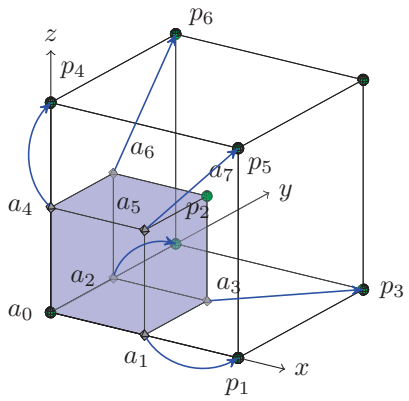


Fig. 1. *Illustration of nodal-connectivities required to perform conforming FEM calculations using a single tree traversal. All the vertices in this illustration are labeled in the Morton ordering. Every octant has at least 2 non-hanging vertices, one of which is shared with the parent and the other is shared amongst all the siblings. The octant shown in blue (a) is a child 0, since it shares its zero vertex (a_0) with its parent (p). It shares vertex a_7 with its siblings. All other vertices, if hanging, point to the corresponding vertex of the parent octant instead. Vertices, a_3, a_5, a_6 are face hanging vertices and point to p_3, p_5, p_6, respectively. Similarly a_1, a_2, a_4 are edge hanging vertices and point to p_1, p_2, p_4.*

standard trilinear shape functions with some minor variations: **(1)** the shape functions are not rooted at the hanging vertices; **(2)** the support of a shape function can spread over more than 8 elements; and **(3)** if a vertex of an element is hanging, then the shape functions rooted at the other non-hanging vertices in that element do not vanish on this hanging vertex. Instead, they will vanish at the non-hanging vertex that this hanging vertex is mapped to. In Figure 1 for example, the shape function rooted at vertex (a_0) will not vanish at vertices a_1, a_2, a_3, a_4, a_5 or a_6; it will vanish at vertices $p_1, p_2, p_3, p_4, p_5, p_6$ and a_7 and assume a value equal to 1 at vertex a_0.

Every octant is owned by a single CPU. The vertices located at the anchors of the octants are owned by the same CPU that owns the octants. However, the values of unknowns associated with octants on inter-CPU boundaries need to be shared among several CPUs. We keep multiple copies of the information related to these octants and we term them "*ghost*" octants. The elements owned by each CPU are categorized into "*independent*" and "*dependent*" elements. While the elements belonging to the latter category require ghost information, the elements belonging to the former category do not. In our implementation of the finite element MatVec, each CPU iterates over all the octants it owns and also loops over a layer of ghost octants that contribute to the vertices it owns.[7] Within the loop, each octant is mapped to one of the 18×8 hanging configurations[8]. This is used to select the appropriate element stencil from a list of pre-computed stencils. We then use the selected stencil in a standard element based assembly technique. Although the CPUs need to read ghost values from other CPUs they only need to

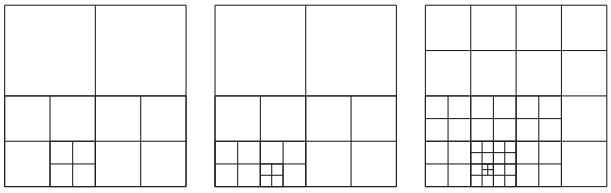
[4]This is not to be confused with the term "*multigrid level*".

[5]This scheme is similar to the one used in [39].

[6]The 2:1 balance constraint ensures that the vertices of the parent can never be hanging.

[7]Each processor only loops over those ghost octants, which are owned by processors with ranks less than its own rank. These ghost octants are referred to as "*pre-ghosts*".

[8]Octants could belong to one of 8 child number types and each child number type is associated with 18 possible hanging vertex configurations.



(a) MG Level $k-2$ (b) MG Level $k-1$ (c) MG Level k

Fig. 2. *Quadtree meshes for three successive multigrid (MG) levels.*

write data back to the vertices they own and do not need to write to ghost vertices. Thus, there is only one communication step within each MatVec. We overlap this communication with computation corresponding to the independent elements.

III. Parallel Geometric Multigrid

The prototypical equation for second-order elliptic operators is the Poisson problem:

$$-\operatorname{div}(\mu(x)\nabla u(x)) = f(x) \quad \forall x \in \Omega, \qquad u(x) = 0 \quad \text{on} \quad \partial\Omega. \tag{1}$$

Here, Ω is the unit cube in three dimensions, $\partial\Omega$ is the boundary of the cube, x is a point in the 3D space, $u(x)$ is the unknown scalar function, $\mu(x)$ is a given scalar function (commonly referred to as a the "*coefficient*" for (1)), and $f(x)$ is a known function. If $\mu(x)$ is smooth, (1) can be efficiently solved with `Dendro`. If μ is discontinuous, we can still use `Dendro` but the convergence rates may be suboptimal.[9] We write the discretized finite-dimensional system of linear equations as $A_k u_k = f_k$, where k denotes the multigrid level.

The `Dendro` *interface:* Let us describe the `Dendro` interface for (1). (The current implementation of `Dendro` supports more general scalar equations, vector equations such as the linear elastostatics equation[10], and time-dependent parabolic, and hyperbolic equations.) The input is μ, f, boundary conditions, and a list of target points in which we want to evaluate the solution. The output is the solution u and, optionally, its gradient at the target points. For simplicity, we assume the following representation for μ: it is approximated as the sum of a constant (background or base value) plus a function that is defined by specifying its value on a set of points. f is represented in a similar manner. We assume that the lists of points that are used to specify μ and f are distributed arbitrarily across CPUs. `Dendro` uses these points to define an octree at the finest multigrid level and subsequently uses the fine level octree to bottom-up construct the multigrid hierarchy. Next, we give details about the main algorithmic components of this procedure.

[9]The current `Dendro` implementation supports isotropic problems only. We are working on extending `Dendro` to anisotropic operators in which $\mu(x)$ is a tensor.

[10]The elastostatics equation is given by $\operatorname{div}(\mu(x)\nabla v(x)) + \nabla(\lambda(x) + \mu(x))\nabla v(x)) = b(x)$, where μ and λ are scalar fields known as Lamé moduli; and v and b are 3D vector functions.

Outline of the `Dendro` *algorithms:* In `Dendro`, we use a hierarchy of octrees (see Figure 2), which we construct as part of the geometric multigrid (GMG) V-cycle algorithm. The V-cycle algorithm consists of 6 main steps: (1) Pre-smoothing: $u_k = S_k(u_k, f_k, A_k)$; (2) Residual computation: $r_k = f_k - A_k u_k$; (3) Restriction: $r_{k-1} = R_k r_k$; (4) Recursion: e_{k-1}=GMG(A_{k-1}, r_{k-1}); (5) Prolongation: $u_k = u_k + P_k e_{k-1}$; and (6) Post-smoothing: $u_k = S_k(u_k, f_k, A_k)$. When "k" reaches the coarsest multigrid level, which we term the "*exact solve*" level, we solve for e_k exactly using a single level solver (e.g., a parallel sparse direct factorization method).

To setup the solver we start with user specified points for μ and f and follow the following steps:

1) Given input points, construct a 2:1 balanced octree corresponding to the finest multigrid level using the algorithms described in [35].
2) Given an upper bound on the total number of multigrid levels, construct the sequence of coarser octrees.
3) Mesh the octrees starting with the octree corresponding to the coarsest multigrid level.
4) Construct restriction and prolongation operators between successive multigrid levels.

In the following sections, we describe the various components of the multigrid algorithm.

A. Global coarsening

Starting with the finest octree, we iteratively construct a hierarchy of complete, balanced, linear octrees such that every octant in the k-th octree (coarse) is either present in the $k+1$-th octree (fine) as well or all its eight children are present instead (Figure 2).

We construct the k-th octree from the $k+1$-th octree by replacing every set of eight siblings by their parent. This is an operation with $\mathcal{O}(N)$ work complexity, where N is the number of leaves in the $k+1$-th octree. It is easy to parallelize and has an $\mathcal{O}(\frac{N}{n_p})$ parallel time complexity, where n_p is the number of CPUs.[11] The main parallel operations are two circular shifts; one clockwise and another anti-clockwise.

However, the operation described above may produce 4:1 balanced octrees[12] instead of 2:1 balanced octrees. Although there is only one level of imbalance that we need to correct, the imbalance can still affect octants that are not in its immediate vicinity. This is known as the *ripple effect*. Even with just one level of imbalance, a ripple can still propagate across many CPUs.

The sequence of octrees constructed as described above has the property that non-hanging vertices in any octree remain non-hanging in all finer octrees as well. Hanging vertices on any octree could either become non-hanging on a finer octree or remain hanging on the finer octrees too. In addition, an octree can have new hanging as well as non-hanging vertices that are not present in any of the coarser octrees.

[11]When we discuss communication costs we assume a Hypercube network topology with $\theta(n_p)$ bandwidth.

[12]The input is 2:1 balanced and we coarsen by at most one level in this operation. Hence, this operation will only introduce one additional level of imbalance resulting in 4:1 balanced octrees.

B. Restriction and Prolongation operators

To implement the intergrid transfer operations, we need to find all the non-hanging fine grid vertices that lie within the support of each coarse grid shape function. These operations can be implemented quite efficiently for the hierarchy of the octree meshes constructed as described in Section III-A. We do not construct these matrices explicitly, instead we implement a matrix-free scheme using MatVecs as described below. The MatVecs for the restriction and prolongation operators are similar.[13] In both the MatVecs, we loop over the coarse and fine grid octants simultaneously. For each coarse-grid octant, the underlying fine-grid octant could either be identical to the coarse octant or one of its eight children. We identify these cases and handle them separately. The main operation within the loop is selecting the coarse-grid shape functions that do not vanish within the current coarse-grid octant and evaluating them at the non-hanging fine-grid vertices that lie within this coarse-grid octant. These form the entries of the restriction and prolongation matrices.

To be able to do this operation efficiently in parallel, we need the coarse and fine grid partitions to be aligned. This means that the following two conditions must be satisfied. **(1)** *If an octant exists both in the coarse and fine grids, then the same CPU must "own" this octant on both the meshes;* and **(2)** *If an octant's children exist in the fine grid, then the same CPU must own this octant on the coarse mesh and all its 8 children on the fine mesh.*

To satisfy these conditions, we first compute the partition on the coarse grid and then impose it on the finer grid. In general, it might not be possible or desirable to use the same partition for all of the multigrid levels because using the same partition for all the levels could cause load imbalance. Also, the coarse levels may be too sparse to be distributed across all of the CPUs. For these reasons, we allow different partitioning across levels. [14] When a transition in the partitions is required, we duplicate the octree at the multigrid level at which the transition between different partitioning takes place and we let one of the duplicates share its partitioning with its immediate finer level and the other one share its partitioning with its immediate coarser level. We refer to one of these duplicates as the"pseudo" mesh (Figure 3). The pseudo mesh is used only to support intergrid transfer operations and smoothing is not performed on this mesh. On these levels, the intergrid transfer operations include an additional step referred to as *Scatter*, which just involves re-distributing the values from one partition to another.

One of the challenges with the MatVec for the intergrid transfer operations is that as we loop over the octants we must also keep track of the pairs of coarse and fine grid vertices that were visited already. In order, to implement this MatVec efficiently we make use of the following observations. **(1)** Every non-hanging fine-grid vertex is shared by at most eight fine-grid elements, excluding the elements whose hanging vertices

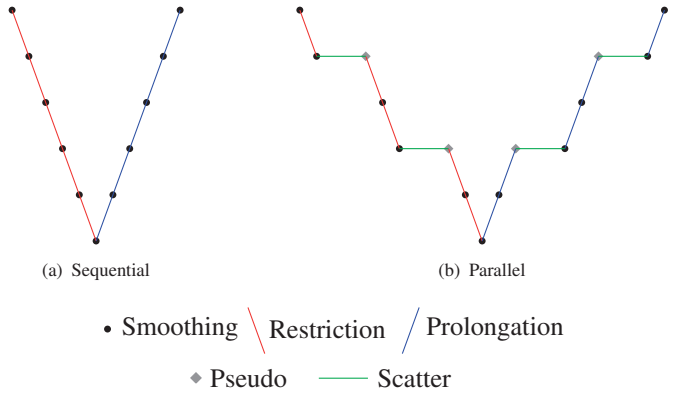


Fig. 3. *(a) A V-cycle where the meshes at all levels share the same partition and (b) A V-cycle in which the multigrid levels do not share the same partition. Some levels do share the same partition and whenever the partition changes a pseudo mesh is added. The pseudo mesh is only used to support intergrid transfer operations and smoothing is not performed on this mesh.*

are mapped to this vertex. **(2)** Each of these eight fine-grid elements will be visited only once within the Restriction and Prolongation MatVecs. **(3)** Since we loop over the coarse and fine grid elements simultaneously, there is a coarse-grid octant[15] associated with each of these eight fine-grid octants. **(4)** The only coarse-grid shape functions that do not vanish at the non-hanging fine-grid vertex under consideration are those whose indices are stored in the vertices of each of these coarse octants. Some of these vertices may be hanging, but they will be mapped to the corresponding non-hanging vertex.

We precompute and store a mask for each fine-grid vertex. Each of these masks is a set of eight bytes, one for each of the eight fine-grid elements that surround this fine-grid vertex. When we visit a fine-grid octant and the corresponding coarse-grid octant within the loop, we read the eight bits corresponding to this fine-grid octant. Each of these bits is a flag that determines whether the corresponding coarse-grid shape function contributes to this fine-grid vertex. The overhead of using this mask within the actual MatVecs includes `(a)` the cost of a few bitwise operations for each fine grid octant and `(b)` the memory bandwidth required for reading the eight-byte mask. The latter cost is comparable to the cost required for reading a material property array within the finite element MatVec (for a variable coefficient operator). Algorithm 1 lists the sequence of operations performed by a CPU for the restriction MatVec. This MatVec is an operation with $\mathcal{O}(N)$ work complexity and has an $\mathcal{O}(\frac{N}{n_p})$ parallel time complexity. For simplicity, we do not overlap communication with computation in the pseudocode. In the actual implementation, we do overlap communication with computation. The following section describes how we compute these masks for any given pair of coarse and fine octrees.

Computing the "masks" for restriction and prolongation: Each non-hanging fine grid vertex has a maximum[16] of 1758 unique locations at which a coarse grid shape function that

[13]The restriction matrix is the transpose of the prolongation matrix.

[14]It is possible that some CPUs are idle on the coarse grids, while no CPU is idle on the finer grids.

[15]This coarse-grid octant is either identical to the fine-grid octant or its parent.

[16]This is a weak upper bound and these cases can be computed offline as they are independent of the octrees under consideration.

Algorithm 1: OPERATIONS PERFORMED BY CPU P FOR THE RESTRICTION MATVEC
Input: Fine vector (F), masks (M), pre-computed stencils (R_1) and (R_2), fine octree (O_f), coarse octree (O_c).
Output: Coarse vector (C).

```
1. Exchange ghost values for F and M with other CPUs.
2. C ← 0.
3. for each o^c ∈ O_c
4.     Let c^c be the child number of o^c.
5.     Let h^c be the hanging type of o^c.
6.     Step through O_f until o^f ∈ O_f is found s.t.
       Anchor(o^f) = Anchor(o^c).
7.     if Level(o^c) = Level(o^f)
8.       for each vertex, V_f, of o^f
9.           Let V_f be the i-th vertex of o^f.
10.          if V_f is not hanging
11.            for each vertex, V_c, of o^c
12.                Let V_c be the j-th vertex of o^c.
13.                If V_c is hanging, use the corresponding
                   non-hanging vertex instead.
14.                if the j-th bit of M(V_f, i) = 1
15.                   C(V_c) = C(V_c) + R_1(c^c, h^c, i, j)F(V_f)
16.                end if
17.            end for
18.          end if
19.      end for
20.    else
21.        for each of the 8 children of o^c
22.            Let c^f be the child number of o^f, the child of o^c
               that is processed in the current iteration.
23.            Perform steps 8 to 19 by replacing R_1(c^c, h^c, i, j)
               with R_2(c^f, c^c, h^c, i, j) in step 15.
24.        end for
25.    end if
26. end for
27. Exchange ghost values for C with other CPUs.
28. Add the contributions received from other CPUs
    to the local copy of C.
```

contributes to this vertex could be rooted. Each of the vertices of the coarse grid octants, which overlap with the fine grid octants surrounding this fine grid vertex, can be mapped to one of these 1758 possibilities. It is also possible that some of these vertices are mapped to the same location. When we pre-compute the masks described earlier, we want to identify these many-to-one mappings and only one of them is selected to contribute to the fine grid vertex under consideration. Details on identifying these cases are given in [32].

We can not pre-compute the masks offline since this depends on the coarse and fine octrees under consideration. To do this computation efficiently, we employ a "dummy" MatVec before we begin solving the problem; this operation is performed only once for each multigrid level. In this dummy MatVec, we use a set of 16 bytes per fine grid vertex; 2 bytes for each of the eight fine grid octants surrounding the vertex. In these 16 bits, we store the flags to determine the following: **(1)** Whether or not the coarse and fine grid octants are the same (1 bit). **(2)** The child number of the current fine grid octant (3 bits).**(3)** The child number of the corresponding coarse grid octant (3 bits). **(4)** The hanging configuration of the corresponding coarse grid octant (5 bits). **(5)** The relative size of the current fine grid octant with respect to the reference element (2 bits). Using this information and some simple bitwise operations, we can compute and store the masks for each fine grid vertex. The dummy MatVec is an operation with $\mathcal{O}(N)$ work complexity and has an $\mathcal{O}(\frac{N}{n_p})$ parallel time complexity.

C. Smoothing and coarse grid operator

We use the standard Jacobi smoothing.[17] For the exact solve we use SuperLU_Dist [25]. The operators A_k are defined using standard FEM discretization. In matrix-free geometric multigrid implementations, the coarse-grid operator is not constructed using the Galerkin condition; instead, a coarse grid discretization

[17] Any read-only-ghost type smoother can be implemented with no additional communication.

of the underlying PDE is used. One can show [32] that these two formulations are equivalent provided the same bilinear form, $a(u, v)$, is used both on the coarse and fine levels. This poses no difficulty for constant coefficient problems or problems in which the material property is described in a closed form. Most often however, the material property is defined just on each fine-grid element. Hence, the bilinear form actually depends on the discretization used. If the coarser grids must use the same bilinear form, the coarser grid MatVecs must be performed by looping over the underlying finest grid elements, using the material property defined on each fine grid element. This would make the coarse grid MatVecs quite expensive. A cheaper alternative would be to define the material properties for the coarser grid elements as the average of those for the underlying fine grid elements. This process amounts to using a different bilinear form for each multigrid level and hence is a clear deviation from the theory. This is one reason why the convergence of the stand-alone multigrid solver deteriorates with increasing contrast in material properties. Coarsening across discontinuities also affects the coarse grid correction, even when the Galerkin condition is satisfied. Large contrasts in material properties also affect simple smoothers like the Jacobi smoother. The standard solution is to use multigrid as a preconditioner to the Conjugate Gradient (CG) method [37]. We have conducted numerical experiments that demonstrate the effectiveness of this approach for the Poisson problem.

D. Metric for load imbalance

In all our MatVecs, the communication cost is proportional to the number of inter-CPU dependent elements and the computation cost is proportional to the total number of local and pre-ghost elements (Section II). We use the following heuristic to estimate the local load on any processor at any given multigrid level:

$$\begin{aligned}\text{Local Load} \;=\; & 0.75 \times (\text{No. of local elements} + \\ & \text{No. of pre-ghost elements}) + \\ & 0.25 \times (\text{No. of dependent elements})\end{aligned} \tag{2}$$

Then, the load imbalance for that multigrid level is computed as the ratio of the maximum and minimum local loads across the processors.

E. Summary of the overall multigrid algorithm

1) A "sufficiently" fine[18] 2:1 balanced complete linear octree is constructed using the algorithms described in [35].
2) Starting with the finest octree, a sequence of 2:1 balanced coarse linear octrees is constructed using the global coarsening algorithm.
3) The maximum number of processors that can be used for each multigrid level without violating the minimum grain size criteria[19] is computed.
4) Starting with the coarsest octree, the octree at each level is meshed using the algorithm described in [34]. As long as the load imbalance (Section III-D) across the CPUs is acceptable and as long as the number of CPUs used for the coarser grid is the same as the maximum number of CPUs that can be used for the finer multigrid level without violating the minimum grain size criteria, the partition of the coarser grid is imposed on to the finer grid during meshing. If either of the above two conditions is violated then the octree for the finer grid is duplicated; One of them is meshed using the partition of the coarser grid and the other is meshed using a fresh partition computed using the "*Block Partition*" algorithm [34], [35]. The process is repeated until the finest octree has been meshed.
5) A dummy restriction MatVec (Section III-B) is performed at each level (except the coarsest) and the masks that will be used in the actual restriction and prolongation MatVecs are computed and stored.
6) For the case of variable coefficient operators, vectors that store the material properties at each level are created.
7) The discrete system of equations is then solved using the conjugate gradient algorithm preconditioned with the multigrid scheme.

Complexity: Let N be the total number of octants in the fine level and n_p be the number of CPUs. In [34] and [35], we showed that the parallel complexity of the single-level construction, 2:1 balancing and meshing is $\mathcal{O}(\frac{N}{n_p}\log\frac{N}{n_p} + n_p \log n_p)$ for trees that are nearly uniform. The cost of a MatVec is $\mathcal{O}(\frac{N}{n_p})$. The complexity of a single-level coarsening (excluding 2:1 balancing) is $\mathcal{O}(\frac{N}{n_p})$.

IV. Adaptive mesh refinement

We solve a linear parabolic problem: $\frac{\partial u}{\partial t} = \Delta u + f(\mathbf{x}, t)$ with homogeneous Neumann boundary conditions using adaptive mesh refinement. We compute $f(\mathbf{x}, \mathbf{t})$ using an analytical solution of a traveling wave of the form $u(\mathbf{x}, t) = \exp(-10^4(y - 0.5 - 0.1t - 0.1\sin(2\pi x))^2)$ (see Figure 6). A second order implicit Crank-Nicholson scheme is used to solve the problem for 10 time steps with a step size of $\delta t = 0.05$. We used CG with Jacobi preconditioner to solve the linear system of discretized equations at every time step. We build the octree using the exact analytical solution. We do not use any error estimator. We focus on demonstrating the performance of our tree-construction, balancing, meshing and solution transfer schemes. Below we describe the **adaptive mesh refinement** scheme.

1) Coarsen or refine octants in an octree using the exact analytical solution at the current time step.
2) Balance the octree to enforce the 2:1 balance constraint.

[18] Here the term sufficiently is used to mean that the discretization error introduced is acceptable.

[19] We require a minimum of 1000 elements per processor for good scalability. If this criteria is not satisfied for any particular multigrid level, then the number of active processors for that level is automatically adjusted. We arrived at this number by assuming an uniform distribution of elements and comparing the number of elements in the interior of a processor and the number of elements on the inter-processor surface. The number of elements in the interior of a processor must be much greater than the number of elements on the inter-processor surface for good scalability.

3) Mesh the octree to get the element-vertex connectivity.
4) Transfer the solution at the previous time step to the new mesh.
5) Solve the linear system of equations.

We also present a parallel algorithm to **map the solution between different multigrid levels** .

1) Get the list of co-ordinates of the vertices in the new mesh $\mathcal{O}(N/n_p)$.
2) Sort the list of co-ordinates using parallel sample sort. $\mathcal{O}(N/n_p \log(N/n_p) + n_p \log n_p)$.
3) Impose the partition of the old mesh on the sorted list $\mathcal{O}(N/n_p)$.
4) Evaluate the function values at the vertices of the new mesh using the nodal values and shape functions of the old mesh $\mathcal{O}(N/n_p)$.
5) Re-distribute the evaluated function values to the partition of the new mesh $\mathcal{O}(N/n_p)$.

The overall computational complexity of the solution transfer algorithm, assuming that the octrees have $\mathcal{O}(N)$ vertices and are similar, is $\mathcal{O}(N/n_p + n_p \log n_p)$. We do not require the meshes at two different time steps to be aligned or to share the same partition. In fact, the two meshes can be entirely different. Of course, the greater their differences the higher the intergrid transfer communication costs.

V. Numerical Experiments

In this section, we report the results from four sets of experiments: (A) Isogranular scalability (weak scaling) for multigrid, (B) Fixed-size scalability (strong scaling) for multigrid, (C) Comparison between Dendro and BoomerAMG and (D) Isogranular scalability for AMR. The experiments were carried out on the Teragrid systems: "Abe" (9.2K Intel-Woodcrest cores with Infiniband), "Bigben" (4.1K AMD Opteron cores with quadrics) and "Ranger" (63K Barcelona cores with Infiniband). Details for these systems can be found in [29], [31], [36]. The following parameters were used in experiments A, B and C: **(1)** A zero initial guess was used, **(2)** `SuperLU_dist` [25] was used as an exact solver at the coarsest grid, **(3)** the forcing term was computed by using a random vector for the exact solution, **(4)** one multigrid V-cycle with 4 pre-smoothing and 4 post-smoothing steps per level was used as a preconditioner to the Conjugate Gradient (CG) method and **(5)** the damped-Jacobi method was used as the smoother at each multigrid level.

Isogranular scalability analysis was performed by roughly keeping the problem size per CPU fixed while increasing the number of CPUs. Fixed-size scalability was performed by keeping the problem size constant while increasing the number of CPUs. In all of the fixed-size and isogranular scalability plots, the first column reports the total setup time, the second column gives the component-wise split-up of the total setup time. The third column represents the total solve time and the last column gives the component-wise split-up for the solve phase. Note that the reported times for each component are the maximum values for that component across all the processors. Hence, in some cases the total time is lower than the sum of the individual components. The setup cost for the GMG scheme (Dendro) includes the time for constructing the mesh for all the levels (including the finest), constructing and balancing all the coarser levels, setting up the intergrid transfer operators by performing one dummy restriction MatVec at each level. The time to create the work vectors for the MG scheme and the time to build the coarsest grid matrix are also included in the total setup time, but are not reported individually since they are insignificant. "Scatter" refers to the process of transferring the vectors between two different partitions of the same multigrid level during the intergrid transfer operations, required whenever the coarse and fine grids do not share the same partition. The time spent in applying the Jacobi preconditioner, computing the inner-products within CG and solving the coarsest grid problems using LU are all accounted for in the total solve time, but are not reported individually since they are insignificant. When we report `MPI_Wait()` times, we refer to synchronization for non-blocking operations during the "Solve" phase.

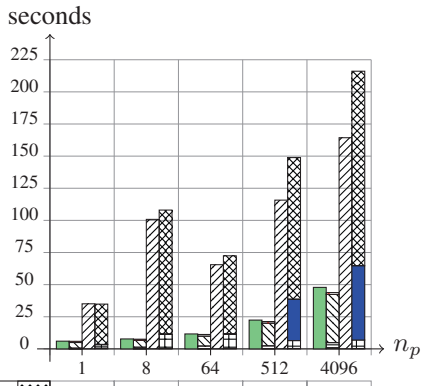
Isogranular scalability for the multigrid solver: In Figure 4(a), we report isogranular scalability results for the constant-coefficient scalar Poisson problem with homogeneous Neumann boundary conditions on meshes whose mesh-size follows a Gaussian distribution. The finest level octrees for the multiple CPU cases were generated using regular refinements from the finest octree for the single CPU case. Hence, the number of elements on the finest multigrid level grows exactly by a factor of 8 every time the number of CPUs is increased by a factor of 8. 4 multigrid levels were used on 1 CPU and the number of multigrid levels were incremented by 1 every time the number of CPUs is increased by a factor of 8. For all the multi-processor runs in this experiment, the coarsest level only used 3 processors.

In Figure 5(a), we report isogranular scalability results for solving the constant-coefficient linear elastostatics problem with homogeneous Dirichlet boundary conditions. Unlike in Figure 4(a), all the octrees used in this experiment were generated directly using Gaussian point distributions of varying number of points. Hence, there is some variation in the grain sizes for the different cases. Notice that the octrees used in this experiment are highly non-uniform (The levels of the coarsest and finest octants at the finest multigrid level are reported in Figure 5(a).). Due to space limitations, we do not report the isogranular scalability results for the Poisson problems (constant and variable coefficient cases) on these octrees; these results are reported in [32].

Fixed-size scalability for the multigrid solver: In Figure 4(b), we report results from the fixed-size scalability experiment for the constant coefficient Poisson problem. The trend is similar for the variable coefficient Poisson problem and the elasticity problem; due to space limitations we do not report these numbers here. More results for different octrees generated using different point distributions are given in [32].

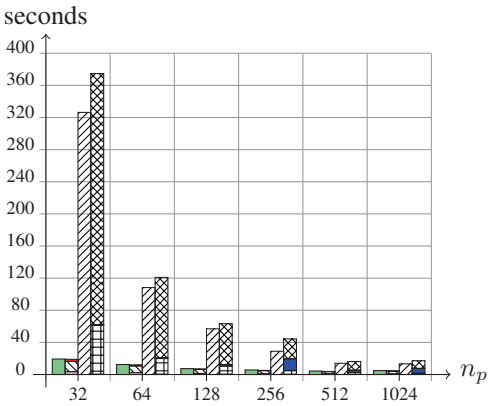
Comparison between Dendro and BoomerAMG: In Figure 5(b), we report results from a comparison with the algebraic multigrid scheme, BoomerAMG.[20] In order to minimize

[20]We did not attempt to optimize our code or AMG and we used the default options.



	1	8	64	512	4096
Coarsening	0.046	0.098	0.099	0.145	0.874
Balancing	0.52	1.19	2.11	2.19	4.0
Meshing	4.39	5.35	7.66	17.65	37.48
R-setup	0.929	0.906	1.34	1.3	1.46
Total Setup	**6.01**	**7.74**	**11.63**	**22.4**	**47.86**
LU	0.539	0.283	0.283	0.294	0.284
R + P	2.89	11.05	11.0	6.16	6.61
Scatter	0	0.215	0.6	32.22	57.81
FE Matvecs	31.41	96.45	60.53	110.16	151.33
Total Solve	**35.07**	**100.76**	**65.5**	**115.73**	**164.29**
Meshes	4	6	9	13	15
MG Levels	4	5	6	7	8
Max/Min Elements	1	1.14	1.77	2.59	2.64
MPI_Wait (sec)	-	49.42	28.14	62.0	93.40

(a) Isogranular Scalability



	32	64	128	256	512	1024
Coarsening	0.201	0.098	0.059	0.048	0.054	0.089
Balancing	3.25	2.28	1.27	0.85	0.71	0.67
Meshing	12.98	8.28	4.96	3.89	2.82	3.42
R-setup	2.50	1.36	0.65	0.36	0.198	0.132
Total Setup	**19.30**	**12.47**	**7.37**	**5.67**	**4.33**	**4.90**
LU	0.282	0.284	0.284	0.285	0.284	0.282
R + P	61.01	19.63	10.2	4.87	2.31	1.32
Scatter	0.905	0.934	1.62	14.46	2.8	5.98
FE Matvecs	312.64	100.1	51.22	24.74	10.76	9.56
Total Solve	**326.44**	**108.3**	**56.89**	**28.93**	**13.97**	**13.26**
Meshes	8	9	9	10	10	11

(b) Fixed-size Scalability

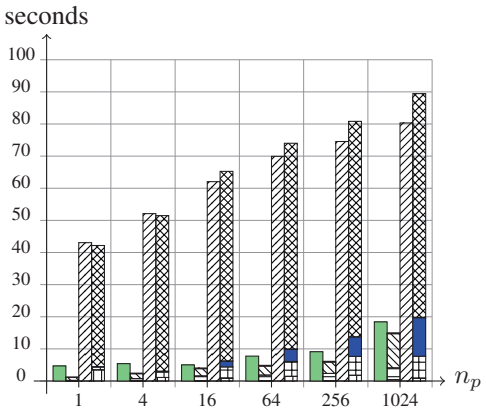
Fig. 4. *LEFT FIGURE: Isogranular scalability for solving the constant coefficient Poisson problem on a set of octrees with a grain size of* $239.4K$ *elements per CPU* (n_p) *on the finest level. The difference between the minimum and maximum levels of the octants on the finest grid is 5. A relative tolerance of* 10^{-10} *in the 2-norm of the residual was used. 6 CG iterations were required in each case, to solve the problem to the specified tolerance. This experiment was performed on "Bigben".*
RIGHT FIGURE: Scalability for a fixed fine grid problem size of 15.3M elements. The problem is the same as described in Figure 4(a). 6 multigrid levels were used. 9 iterations were required to solve the problem to a relative tolerance of 10^{-14} *in the 2-norm of the residual. This experiment was performed on "Bigben".*

communication costs, the coarsest level was distributed on a maximum of 32 CPUs in all experiments. For BoomerAMG, we experimented with two different coarsening schemes: Falgout coarsening and CLJP coarsening. The results from both experiments are reported. Falgout coarsening works best for structured grids and CLJP coarsening is better suited for unstructured grids [22]. We compare our results using both the schemes. Both GMG and AMG schemes used 4 pre-smoothing steps and 4 post-smoothing steps per level with the damped Jacobi smoother. A relative tolerance of 10^{-10} in the 2-norm of the residual was used in all the cases. The GMG scheme took about 12 CG iterations, the Falgout scheme took about 7 iterations and the CLJP scheme also took about 7 iterations.

The setup time reported for the AMG schemes includes the time for meshing the finest grid and constructing the finest grid FE matrix, both of which are quite small (≈ 1.35 seconds for meshing and ≈ 22.93 seconds for building the fine grid matrix even on 256 CPUs) compared to the time to setup the rest of the AMG scheme. Although GMG seems to be performing well, more difficult problems with multiple discontinuous coefficients can cause it to fail. *Our method is not robust in the presence of discontinuous coefficients—in contrast to AMG.*

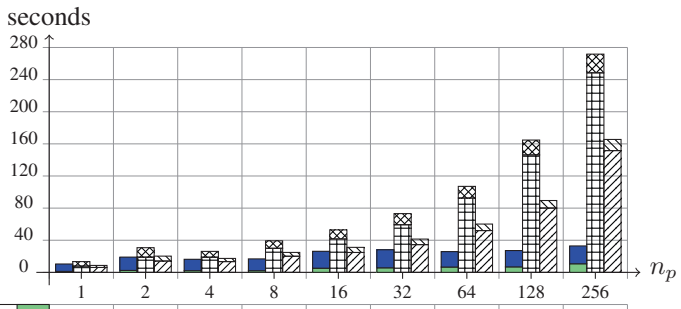
Discussion on scalability results: The setup costs are much smaller than the solution costs, suggesting that `Dendro`is suitable for problems that require the construction and solution of linear systems of equations numerous times. The increase in running times for the large processor cases can be primarily attributed to poor load balancing. This can be seen from **(a)** the total time spent in calls to the `MPI_Wait()` function and **(b)** the imbalance in the number of elements (own + pre-ghosts) per processor. These numbers are reported in Figure 4(a) and Figure 5(a).[21] Load balancing is a challenging problem due to the following reasons: **(1)** We need to make an accurate a-priori estimate of the computation and communication loads–it is difficult to make such estimates for arbitrary distributions; **(2)** for the intergrid transfer operations, the coarse and fine grids need to be aligned-it is difficult to get good load balance for both the grids, especially for non-uniform distributions; and **(3)** Partioning each multigrid level independently to get good load balance for the smoothing operations at each multigrid level would require the creation of an auxiliary mesh for each multigrid level and a scatter operation for each intergrid transfer

[21] We only report the Max/Min elements ratios for the finest multigrid level although the trend is similar for other levels as well.



		1	4	16	64	256	1024
Coarsening		0.008	0.019	0.02	0.026	0.042	0.427
Balancing		0.178	0.712	1.54	1.81	2.31	3.65
Meshing		0.913	1.54	2.28	2.81	3.53	10.65
R-setup		0.146	0.184	0.219	0.232	0.25	0.279
Total Setup		**4.69**	**5.41**	**5.02**	**7.75**	**9.12**	**18.41**
LU		3.44	1.1	0.85	1.45	1.78	0.84
R + P		0.94	1.68	3.54	4.64	5.89	6.88
Scatter		0	0.245	1.83	3.82	6.06	11.99
FE Matvecs		37.80	48.46	59.03	64.09	67.09	69.74
Total Solve		**43.06**	**52.1**	**62.05**	**69.93**	**74.55**	**80.32**
Meshes		3	6	8	10	12	20
MG Levels		3	5	6	7	8	10
Elements		30.9K	114K	447.5K	1.77M	7.04M	28.23M
Max/Min Elements		1	1.21	1.91	2.44	2.78	2.56
MPI_Wait (sec)		-	6.73	25.58	35.50	41.17	40.56
Finest Octant's Level		10	11	12	15	16	18

(a) Isogranular Scalability for Dendro



		1	2	4	8	16	32	64	128	256
GMG Setup		1.21	2.42	2.11	2.05	5.19	5.67	6.63	6.84	10.7
GMG Solve		9.23	16.6	14.19	14.76	21.07	22.49	19.2	20.33	22.23
Falgout Setup		7.86	18.97	18.9	30	41.56	59.29	92.61	146.23	248.5
Falgout Solve		5.54	11.77	7.23	9.25	11.35	13.91	14.5	18.56	23.3
CLJP Setup		6.09	14.12	13.41	20.08	24.77	34.54	52.05	80.09	151.64
CLJP Solve		2.56	6.11	3.96	4.77	6.38	6.92	8.12	9.4	13.9
Elements		52K	138K	240K	0.5M	995K	2M	3.97M	8M	16M

(b) Dendro Vs BoomerAMG

Fig. 5. *TOP FIGURE: Isogranular scalability for solving the constant coefficient linear elastostatics problem on a set of octrees with a grain size of* $30K$ *(approx) elements per CPU* (n_p) *on the finest level. In each case, the coarsest octant at the finest multigrid level was at level 3; the level of the finest octant at the finest multigrid level is reported in the figure, and can go up to 18. Thus, the octrees considered here are extremely non-uniform–roughly five-orders of magnitude variation in the leaf size. A relative tolerance of* 10^{-8} *in the 2-norm of the residual was used. 9 CG iterations were required in each case, to solve the problem to the specified tolerance. This experiment was performed on "Ranger".*
BOTTOM FIGURE: Comparison between Dendro (GMG) and BoomerAMG for solving a variable coefficient (contrast of 10^7*) scalar elliptic problem with homogeneous Neumann boundary conditions on meshes constructed on octrees generated using Gaussian distributions. For BoomerAMG, we experimented with two different coarsening schemes: Falgout coarsening and CLJP coarsening. This experiment was performed on "Abe". Each node of the cluster has 8 CPUs which share an 8GB RAM. However, only 1 CPU per node was utilized in the above experiments. This is because the AMG scheme required a lot of memory and this allowed the entire memory on any node to be available for a single process. We caution the reader that we have not attempted to optimize the AMG solver and that the jumping-coefficient problem we consider here is relatively easy since it has only one discontinuity "island".*

n_p	Elements	iter	Solve	Balancing	Meshing	Transfer
8	300K	110	78.1	6.24	6.0	0.76
64	2M	204	143.9	17.0	7.86	0.82
512	14M	384	356	132.9	72.3	3.32

TABLE I

*Isogranular scalability with a grain size of $38K$ (approx) elements per CPU (n_p). A constant-coefficient linear parabolic problem with homogeneous Neumann boundary conditions was solved on octree meshes. We used the analytical solution of a traveling wave of the form $\exp(-10^4(y-0.5-0.1t-0.1\sin(2\pi x))^2)$ to construct the octrees. We used a time step of $\delta t = 0.05$ and solved for 10 time steps. We used second-order Crank-Nicholson scheme for time-stepping and CG with Jacobi preconditioner to solve the linear system of equations at every time step. We used a stopping criterion of $\|r\|/\|r_0\| < 10^{-6}$. We report the number of elements (***Elements***), CG iterations (**iter**), Solve time (**Solve**), total time to balance (**Balancing**) and mesh (**Meshing**) the octrees generated at each time step. We also report the time to transfer the solution between the meshes (**Transfer**) at two different time steps. The total number of elements and the number of CG iterations are approximately constant over all time steps. This experiment was performed on "Abe".*

operation at each multigrid level. This would increase the setup and communication costs.

Adaptive mesh refinement: Finally, in Table I and Figure 6, we report the performance of the balancing, meshing and the solution transfer algorithms to solve the linear parabolic problem described in Section IV.

VI. Conclusions

We have described a parallel matrix-free geometric multiplicative multigrid method for solving second order elliptic partial differential equations using finite elements on octrees. Also, we described a scheme that can be used to transfer scalar and vector functions between arbitrary octrees during adaptive mesh refinement.

We automatically generate a sequence of coarse meshes from an arbitrary 2:1 balanced fine octree. We do not impose any restrictions on the number of meshes in this sequence or the size of the coarsest mesh. We do not require the meshes to be aligned and hence the different meshes can be partitioned independently. Although a bottom-up tree construction and meshing is harder than top-down approaches, we believe that it is more practical since the fine mesh can be defined naturally based on the physics of the problem.

We have demonstrated the scalability of our implementation and can solve problems with billions of elements on thousands of CPUs. We have demonstrated that our implementation works well even on problems with largely varying material properties. We have compared our geometric multigrid implementation (Dendro) with a state-of-the-art algebraic multigrid implementation (BoomerAMG) in a standard off-the-shelf package (HYPRE); Dendro has a much lower setup cost compared to BoomerAMG and hence is better suited for adaptive mesh refinement procedures. Overall, we showed that the proposed algorithm is quite efficient although significant work remains to improve the partitioning algorithm and the overall robustness of the scheme in the presence of discontinuous coefficients.

Acknowledgments.

This work was supported by the U.S. Department of Energy under grant DE-FG02-04ER25646, and the U.S. National Science Foundation grants CCF-0427985, CNS-0540372, DMS-0612578, OCI-0749285 and OCI-0749334. Computing resources on the TeraGrid systems were provided under the grants ASC070050N and MCA04T026. We would like to thank the TeraGrid support staff, and staff and consultants at NCSA, PSC and TACC. We would also like to thank the reviewers for their constructive criticisms.

References

[1] M.F. Adams, H.H. Bayraktar, T.M. Keaveny, and P. Papadopoulos. Ultrascalable implicit finite element analyses in solid mechanics with over a half a billion degrees of freedom. In *Proceedings of SC2004*, The SCxy Conference series in high performance networking and computing, Pittsburgh, Pennsylvania, 2004. ACM/IEEE.

[2] Volkan Akcelik, Jacobo Bielak, George Biros, Ioannis Epanomeritakis, Antonio Fernandez, Omar Ghattas, Eui Joong Kim, Julio Lopez, David R. O'Hallaron, Tiankai Tu, and John Urbanic. High resolution forward and inverse earthquake modeling on terascale computers. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. ACM, 2003.

[3] WK Anderson, WD Gropp, DK Kaushik, DE Keyes, and BF Smith. Achieving high sustained performance in an unstructured mesh CFD application. *Supercomputing, ACM/IEEE 1999 Conference*, 1999.

[4] Satish Balay, Kris Buschelman, William D. Gropp, Dinesh Kaushik, Matt Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc home page, 2001. www.mcs.anl.gov/petsc.

[5] Gregory T. Balls, Scott B. Baden, and Phillip Colella. SCALLOP: A highly scalable parallel Poisson solver in three dimensions. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.

[6] R. Becker and M. Braack. Multigrid techniques for finite elements on locally refined meshes. *Numerical Linear Algebra with applications*, 7:363–379, 2000.

[7] B Bergen, F. Hulsemann, and U. Rude. Is 1.7×10^{10} Unknowns the Largest Finite Element System that Can Be Solved Today? In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 5, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Marshall W. Bern, David Eppstein, and Shang-Hua Teng. Parallel construction of quadtrees and quality triangulations. *International Journal of Computational Geometry and Applications*, 9(6):517–532, 1999.

[9] M. Bittencourt and R. Feij'oo. Non-nested multigrid methods in finite element linear structural analysis. In *Virtual Proceedings of the 8th Copper Mountain Conference on Multigrid Methods (MGNET)*, 1997.

[10] Hans-Joachim Bungartz, Miriam Mehl, and Tobias Weinzierl. A parallel adaptive cartesian pde solver using space–filling curves. In E. Wolfgang Nagel, V. Wolfgang Walter, and Wolfgang Lehner, editors, *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference*, volume 4128 of *LNCS*, pages 1064–1074, Berlin Heidelberg, 2006. Springer-Verlag.

[11] Paul M. Campbell, Karen D. Devine, Joseph E. Flaherty, Luis G. Gervasio, and James D. Teresco. Dynamic octree load balancing using space-filling curves. Technical Report CS-03-01, Williams College Department of Computer Science, 2003.

[12] Edmond Chow, Robert D. Falgout, Jonathan J. Hu, Raymond S. Tuminaro, and Ulrike M. Yang. A survey of parallelization techniques for multigrid solvers. In Michael A. Heroux, Padma Raghavan, and Horst D. Simon, editors, *Parallel Processing for Scientific Computing*, pages 179–195. Cambridge University Press, 2006.

[13] William M. Deen. *Analysis of transport phenomena*. Topics in Chemical Engineering. Oxford University Press, New York, 1998.

[14] L.F. Demkowicz, J.T. Oden, and W. Rachowicz. Toward a universal h-p adaptive finite element strategy part 1: Constrained approximation and data structure. *Computer Methods in Applied Mechanics and Engineering*, 77:79–112, 1989.

[15] R.D. Falgout, J.E. Jones, and U.M. Yang. The design and implementation of Hypre, a library of parallel high performance preconditioners. In A.M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51, pages 267–294. Springer-Verlag, 2006.

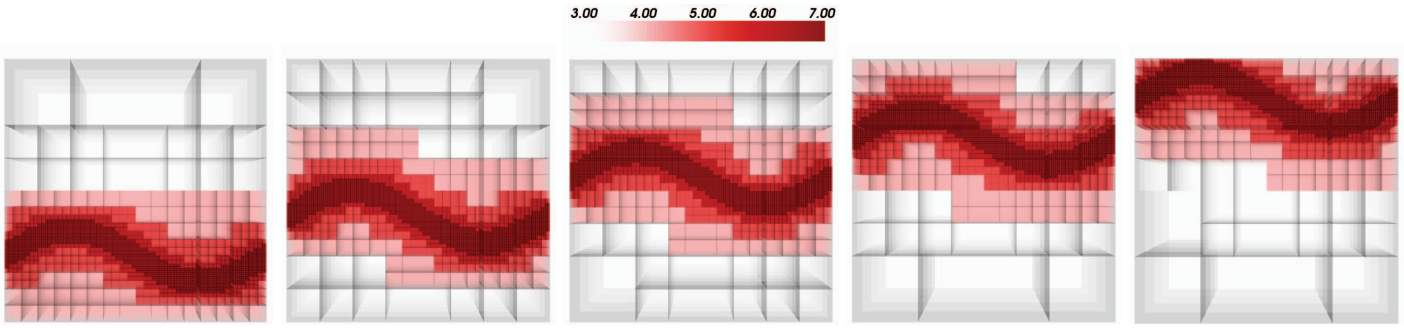


Fig. 6. *An example of adaptive mesh refinement on a traveling wave. This is a synthetic solution which we use to illustrate the adaptive octrees. The coarsening and refinement are based on the approximation error between the discretized and the exact function.*

[16] Robert Falgout. An introduction to algebraic multigrid. *Computing in Science and Engineering, Special issue on Multigrid Computing*, 8:24–33, 2006.

[17] D. M. Greaves and A. G. L. Borthwick. Hierarchical tree-based finite element mesh generation. *International Journal for Numerical Methods in Engineering*, 45(4):447–471, 1999.

[18] M. Griebel and G. Zumbusch. Parallel multigrid in an adaptive PDE solver based on hashing. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and U. Trottenberg, editors, *Parallel Computing: Fundamentals, Applications and New Directions, Proceedings of the Conference ParCo'97, 19-22 September 1997, Bonn, Germany*, volume 12, pages 589–600, Amsterdam, 1998. Elsevier, North-Holland.

[19] W.D. Gropp, D.K. Kaushik, D.E. Keyes, and B.F. Smith. Performance modeling and tuning of an unstructured mesh CFD application. *Proc. SC2000: High Performance Networking and Computing Conf.(electronic publication)*, 2000.

[20] Frank Günther, Miriam Mehl, Markus Pögl, and Christoph Zenger. A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves. *SIAM Journal on Scientific Computing*, 28(5):1634–1650, 2006.

[21] Morton E. Gurtin. *An introduction to continuum mechanics*, volume 158 of *Mathematics in Science and Engineering*. Academic Press, San Diego, 2003.

[22] Van Emden Henson and Ulrike Meier Yang. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41(1):155–177, 2002.

[23] Frank H ulsemann, Markus Kowarschik, Marcus Mohr, and Ulrich R ude. Parallel geometric multigrid. In Are M. Bruaset and Aslka Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, pages 165–208. Birk auser, 2006.

[24] A.C. Jones and P.K. Jimack. An adaptive multigrid tool for elliptic and parabolic systems. *International Journal for Numerical Methods in Fluids*, 47:1123–1128, 2005.

[25] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

[26] Dimitri J. Mavriplis, Michael J. Aftosmis, and Marsha Berger. High resolution aerospace applications using the NASA Columbia Supercomputer. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 61, Washington, DC, USA, 2005. IEEE Computer Society.

[27] Miriam Mehl. Cache-optimal data-structures for hierarchical methods on adaptively refined space-partitioning grids, September 2006.

[28] Miriam Mehl, Tobias Weinzierl, and Christoph Zenger. A cache-oblivious self-adaptive full multigrid method. *Numerical Linear Algebra with Applications*, 13(2-3):275–291, 2006.

[29] NCSA. Abe's system architecture. ncsa.uiuc.edu/UserInfo/Resources/Hardware/Intel64Cluster.

[30] S. Popinet. Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries. *Journal of Computational Physics*, 190:572–600(29), 20 September 2003.

[31] PSC. Bigben's system architecture. www.psc.edu/machines/cray/xt3.

[32] Rahul Sampath and George Biros. A parallel geometric multigrid method for finite elements on octree meshes. Technical report, Georgia Institute of Technology, 2008. Submitted for publication.

[33] Rahul Sampath, Hari Sundar, Santi S. Adavani, Ilya Lashuk, and George Biros. `Dendro` home page, 2008. www.cc.gatech.edu/csela/dendro.

[34] Hari Sundar, Rahul Sampath, Santi S. Adavani, Christos Davatzikos, and George Biros. Low-constant parallel algorithms for finite element simulations using linear octrees. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, The SCxy Conference series in high performance networking and computing, Reno, Nevada, 2007. ACM/IEEE.

[35] Hari Sundar, Rahul S. Sampath, and George Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708, 2008.

[36] TACC. Ranger's system architecture. www.tacc.utexas.edu/resources/hpcsystems.

[37] Trottenberg, U. and Oosterlee, C. W. and Schuller, A. *Multigrid*. Academic Press Inc., San Diego, CA, 2001.

[38] Tiankai Tu, David R. O'Hallaron, and Omar Ghattas. Scalable parallel octree meshing for terascale applications. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 4, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Weigang Wang. Special bilinear quadrilateral elements for locally refined finite element grids. *SIAM Journal on Scientific Computing*, 22:2029–2050, 2001.

[40] Brian S. White, Sally A. McKee, Bronis R. de Supinski, Brian Miller, Daniel Quinlan, and Martin Schulz. Improving the computational intensity of unstructured mesh applications. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 341–350, New York, NY, USA, 2005. ACM Press.