A PARALLEL GEOMETRIC MULTIGRID METHOD FOR FINITE ELEMENTS ON OCTREE MESHES

RAHUL S. SAMPATH* AND GEORGE BIROS†

Abstract. In this article, we present a parallel geometric multigrid algorithm for solving elliptic partial differential equations (PDEs) on octree based conforming finite element discretizations. We describe an algorithm for constructing the coarser multigrid levels starting with an arbitrary 2:1 balanced fine-grid octree discretization. We also describe matrix-free implementations for the discretized finite element operators and the intergrid transfer operations. The key component of our scheme is an octree meshing algorithm, which handles "*hanging*" vertices in a manner that naturally supports conforming trilinear shape functions. Our MPI-based implementation has scaled to billions of elements on thousands of processors on the Cray XT3 MPP system "*Bigben*" at the Pittsburgh Supercomputing Center (PSC) and the Intel 64 Linux Cluster "*Abe*" at the National Center for Supercomputing Applications (NCSA). Although we do not discuss adaptive mesh refinement here, the proposed method can be used efficiently in such problems since it has a low setup cost.

Key words. Geometric Multigrid, Finite Element Method, Linear Octrees, Adaptive Meshes, Matrix-Free Methods, Iterative Solvers, Parallel Algorithms, Tree Codes

AMS subject classifications. 65N30, 65N50, 65N55, 65Y05, 68W10, 68W15

1. Introduction.

Various physical and biological processes are modelled using elliptic operators, such as the Laplacian operator. They are often encountered in heat and mass transfer theory [20], solid and fluid mechanics [20, 30], electromagnetism [27], quantum mechanics [28], models for tumor growth [4], protein folding and binding [42] and cardiac electrophysiology [44]. They are also used in non-physical applications such as mesh generation [49], image segmentation [24] and image registration [41].

The finite element method is a popular technique for numerically solving such partial differential equations over complex domains and/or domains with moving boundaries and for adaptive applications. Structured grids are seldom suited for these applications due to their limited flexibility. However, the flexibility of unstructured meshes comes at a price – they incur the overhead of explicitly constructing element-to-node connectivity information, are unsuitable for matrix-free implementations and are generally cache inefficient because of random queries into this data structure [5, 29, 58]. Octree meshes seem like a promising alternative, at least for some problems [3, 9, 43]; they are more flexible than structured grids, the overhead of constructing element-to-node connectivity information is lower than that of unstructured grids, they allow for matrix-free implementations and the cost of applying the discretized Laplacian operator with octree discretizations is comparable to that of a discretization on a regular grid with the same number of elements [50].

Multigrid methods for solving such partial differential equations (PDEs) have been researched extensively in the last two decades [8, 13, 14, 21, 31, 32, 48, 59, 60, 61, 62] and continue to remain an active area of research [1, 2, 9, 10, 26, 31, 35]. There are numerous works on the theoretical and practical aspects of the different multigrid schemes (V-cycle, W-cycle, FMV-cycle) for a variety of meshes ranging from simple structured grids to non-nested unstructured meshes. A distinguishing

*Department of Mechanical Engineering and Applied Mechanics, University of Pennsylvania, Philadelphia, PA-19104 (rahulss@seas.upenn.edu).

†Departments of Mechanical Engineering and Applied Mechanics, Bioengineering and Computer and Information Science, University of Pennsylvania, 220 S, 33rd Street, Philadelphia, PA, 19104-6315 (biros@seas.upenn.edu).

feature of multigrid schemes is that their convergence rates do not deteriorate with increasing problem size. Moreover, they have optimal complexity for solving certain types of problems [17, 32, 54].

Multigrid algorithms can be classified into two categories: (a) Geometric and (b) Algebraic; The primary difference being that the algorithms of the former type use an underlying mesh for constructing coarser levels ("*coarsening*") and the algorithms of the latter type use the entries of the fine-grid matrix for coarsening. Algebraic multigrid methods are gaining prominence due to their generality and the ability to deal with unstructured meshes. In contrast, geometric multigrid methods are less general. However, in situations where geometric multigrid methods work they have low overhead, are quite fast and, when the grid is cartesian, are easy to parallelize. For this reason, geometric multigrid methods have been quite popular for solving smooth coefficient non-oscillatory elliptic PDEs on nearly structured meshes.

The major hurdles with implementing geometric multigrid methods on unstructured meshes are coarsening and the construction of appropriate intergrid transfer operations. In this work, we show how the use of octrees instead of generic unstructured meshes can alleviate some of these issues. Our parallel geometric multigrid implementation is built on top of the octree data structures developed in our recent work [50, 51].

Related Work. There is a vast literature on multigrid methods for solving partial differential equations (PDEs). Here, we only review some of the recent work on adaptive meshes. In [12], a sequential geometric multigrid algorithm was used to solve two and three dimensional linear elastic problems using finite elements on non-nested unstructured triangular and tetrahedral meshes, respectively. The implementation of the intergrid transfer operations described in this work can be quite expensive for large problems and is non-trivial to parallelize. A sequential full approximation multigrid scheme for finite element simulations of non-linear problems on quadtree meshes was described in [35]. In addition to the 2:1 balance[1] constraint, a specified number of "*safety layers*" of octants were added at each multigrid level to support their intergrid transfer operations. Projections were also required at each multigrid level to preserve the continuity of the solution, which is otherwise not guaranteed using their non-conforming discretizations. Projection schemes require two additional tree-traversals per MatVec, which we avoid in our approach. A 3-D parallel algebraic multigrid method for unstructured finite element problems was presented in [2]. In this work, the authors used parallel maximal independent set algorithms for constructing the coarser grids and constructed the Galerkin coarse-grid operators algebraically using the restriction operators and the fine-grid operator. In [10], a calculation with over 11 billion elements was reported. The authors proposed a scheme for conforming discretizations and geometric multigrid solvers on semi-structured meshes. That approach is highly scalable for nearly structured meshes and for constant coefficient PDEs. However, it limits adaptivity because it is based on regular refinement. Moreover, its computational efficiency diminishes in the case of variable-coefficient operators. Additional examples of scalable approaches for unstructured meshes include [1] and [40]. In those works, multigrid approaches for general elliptic operators were proposed. The associated constants for constructing the mesh and performing the calculations however, are quite large. A significant part of CPU time is related to the multigrid scheme. The high-costs related to partitioning, setup, and accessing generic unstructured grids, has motivated the design of octree-based data structures.

[1] A formal definition of the 2:1 balance constraint is given in Definition 1.

Such constructions have been used in sequential and modestly parallel adaptive finite element implementations [9, 25, 43]. A characteristic of octree meshes is that they contain "*hanging*" vertices. In [50], 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 processors on a Cray XT3 at the Pittsburgh Supercomputing Center. We also showed that the cost of applying the Laplacian operator using this framework is comparable to that of applying it using a direct indexing regular grid discretization with the same number of elements.

Contributions. Although several sequential and parallel implementations for both geometric and algebraic multigrid methods are available [2, 7, 33], to our knowledge there is no work on parallel, octree-based, geometric multigrid solvers for finite element discretizations. In this work, we propose a parallel bottom-up geometric multigrid algorithm on top of the 2:1 balancing and meshing algorithms [50, 51] that were developed in our group. Also, we conducted numerical experiments that demonstrate the effectiveness of the method. In designing the new algorithms, our goals have been minimization of memory footprint, low setup costs, and end-to-end[2] parallel scalability:

- We propose a parallel global coarsening algorithm to construct a series of 2:1 balanced coarser octrees and corresponding meshes starting with an arbitrary 2:1 balanced fine octree. We do not impose any restrictions on the number of multigrid levels or the size of the coarsest mesh. Global coarsening poses difficulties with partitioning and load balancing due to the fact that even if the input to the coarsening function is load balanced, the output may not be so.
- Transferring information between succesive multigrid levels in parallel is a challenging task because the coarse and fine grids may have been partitioned across processors in a completely different way. In the present work, we describe a scalable, matrix-free implementation of the intergrid transfer operators.
- The MPI-based implementation of our multigrid method, DENDRO, has scaled to billions of elements on thousands of processors even for problems with large contrasts in the material properties. Dendro is an open source code that can be downloaded from [46]. Dendro is tightly integrated with PETSc [7].

Limitations. Some of the limitiations of the proposed methodology are listed below:

- Our current implementation results in a second order accurate method. A higher order method can be obtained either by extending [50] to support higher order discretizations or by using an extrapolation technique such as the one suggested in [36].
- Problems with complex geometries are not directly supported in our implementation; in principle, Dendro can be combined with fictitious domain methods [23, 45] to allow solution of such problems but the computational costs will increase and the order of accuracy will be reduced.
- The method is not robust for problems with large jumps in the material properties. However, we do observe good results for such problems in our experiments.

[2]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.

- We observe some problems with the load balancing across processors as we move to a large number of processors.

Organization of the paper. In Section 2, we present a symmetric variational problem and describe a V-cycle multigrid algorithm to solve the corresponding discretized system of equations. It is common to work with discrete, mesh-dependent, inner products in these derivations so that inverting the Gram matrix[3] can be avoided [8, 14, 15, 16, 60, 61, 62]. However, we do not impose any such restrictions. Instead, we show (Section 2.5) how to avoid inverting the Gram matrix for any choice of the inner-product. In Section 3, we describe a matrix-free implementation for the multigrid method. In particular, we describe our framework for handling hanging vertices[4] and how we use it to implement the MatVecs[5] for the finite element matrices as well as the restriction/prolongation matrices. In Section 4, we present the results from fixed-size and iso-granular scalability experiments. We also compare our implementation with "*BoomerAMG*" [33], an algebraic multigrid implementation available in the parallel linear algebra package "*Hypre*" [22]. In Section 5, we present the conclusions from this study and also provide some suggestions for future work.

2. A finite element multigrid formulation.

2.1. Variational problem. Given a domain $\Omega \subset \mathcal{R}^3$ and a bounded, symmetric bilinear form, $a(u, v)$, that is coercive on $H^1(\Omega)$ and $f \in L^2(\Omega)$, we want to find $u \in H^1(\Omega)$ such that u satisfies

$$a(u, v) = (f, v)_{L^2(\Omega)} \quad \forall v \in H^1(\Omega) \tag{2.1}$$

and the appropriate boundary conditions on the boundary of the domain, $\partial\Omega$. This problem has a unique solution [16].

2.1.1. Galerkin approximation. In this section, we derive a discrete set of equations that need to be solved to find an approximate solution for Equation 2.1. First, we define a sequence of nested *finite* dimensional spaces, $V_1 \subset V_2 \subset \cdots \subset H^1(\Omega)$, all of which are subspaces of $H^1(\Omega)$. Here, V_k corresponds to a fine mesh and V_{k-1} corresponds to the immediately coarser mesh. The discretized problem is then to find an approximation of u, $u_k \in V_k$, such that

$$a(u_k, v) = (f, v)_{L^2(\Omega)} \quad \forall v \in V_k. \tag{2.2}$$

The discretized problem has a unique solution and the sequence $\{u_k\}$ converges to u [16].

Let $(\cdot, \cdot)_k$ be an inner-product defined on V_k. By using the linear operator $A_k : V_k \to V_k$ defined by

$$(A_k v, w)_k = a(v, w) \quad \forall v, w \in V_k, \tag{2.3}$$

the discretized problem can be restated as follows: Find $u_k \in V_k$, which satisifies

$$A_k u_k = f_k \tag{2.4}$$

[3] Given an inner-product and a set of vectors, the Gram matrix is defined as the matrix whose entries are the inner-products of the vectors.

[4] This work first appeard in [50]; Here we present it in much greater detail.

[5] A MatVec is a function that takes a vector as input and returns another vector, the result of applying the operator on the input vector.

where $f_k \in V_k$ is defined by

$$(f_k, v)_k = (f, v)_{L^2(\Omega)} \quad \forall v \in V_k \tag{2.5}$$

Appendix A shows that the operator A_k is a symmetric (self-adjoint) positive operator w.r.t $(\cdot,\cdot)_k$. In the following sections, we use italics to represent an operator (or vector) in the continuous form and use bold face to represent the matrix (or vector) corresponding to its co-ordinate basis representation.

Let $\left\{\phi_1^k, \phi_2^k, \ldots, \phi_{dim(V_k)}^k\right\}$ be a basis for V_k. Then, we can show the following:

$$\begin{aligned}
\mathbf{A_k} &= (\mathbf{M_k^k})^{-1}\tilde{\mathbf{A}}_\mathbf{k} \\
\mathbf{f_k} &= (\mathbf{M_k^k})^{-1}\tilde{\mathbf{f}}_\mathbf{k} \\
\mathbf{M_k^k}(i,j) &= (\phi_i^k, \phi_j^k)_k \\
\tilde{\mathbf{A}}_\mathbf{k}(i,j) &= a(\phi_i^k, \phi_j^k) \quad \forall i,j = 1,2,\ldots,dim(V_k) \\
\tilde{\mathbf{f}}_\mathbf{k}(j) &= (f, \phi_j^k)_{L^2(\Omega)} \quad \forall j = 1,2,\ldots,dim(V_k)
\end{aligned} \tag{2.6}$$

In Equation 2.6, $\mathbf{M_k^k}$ is the Gram or mass matrix.

2.2. Prolongation. The prolongation operator is a linear operator

$$P : V_{k-1} \rightarrow V_k \tag{2.7}$$

defined by

$$Pv = v \quad \forall v \in V_{k-1} \subset V_k. \tag{2.8}$$

This is a standard prolongation operator and has been used before [16, 17]. The variational form of Equation 2.8 is given by

$$(Pv, w)_k = (v, w)_k \quad \forall v \in V_{k-1}\ ,\ w \in V_k. \tag{2.9}$$

In Appendix B, we show that

$$\mathbf{P}(i,j) = \phi_j^{k-1}(p_i). \tag{2.10}$$

In equation 2.10, p_i is the fine-grid vertex associated with the fine-grid finite element shape function, ϕ_i^k and ϕ_j^{k-1} is a coarse-grid finite element shape function.

2.3. Coarse-grid problem. The coarse-grid problem can be stated as follows: Find $v_{k-1} \in V_{k-1}$ that satisfies

$$A_{k-1}^G v_{k-1} = f_{k-1}^G \tag{2.11}$$

where, A_{k-1}^G and f_{k-1}^G are defined by the "*Galerkin*" condition (Equation 2.12) [17].

$$\boxed{\begin{aligned} A^G_{k-1} &= P^* A_k P \\ f^G_{k-1} &= P^*(A_k v_k - f_k), \\ &\qquad \forall v_{k-1} \in V_{k-1}, v_k \in V_k \end{aligned}} \tag{2.12}$$

Here, P is the prolongation operator defined in Section 2.2 and P^* is the Hilbert adjoint operator[6] of P with respect to the inner-products $(\cdot,\cdot)_k$ and $(\cdot,\cdot)_{k-1}$. The derivation of the Galerkin condition is given in Appendix C.

2.4. Restriction. Since the restriction operator must be the Hilbert adjoint of the prolongation operator, we define the restriction operator $R : V_k \to V_{k-1}$ as follows:

$$(Rw, v)_{k-1} = (w, Pv)_k = (w, v)_k \quad \forall v \in V_{k-1}, w \in V_k \tag{2.13}$$

In Appendix D, we show that

$$\boxed{\mathbf{R} = (\mathbf{M^{k-1}_{k-1}})^{-1}\mathbf{M^{k-1}_k}} \tag{2.14}$$

where,

$$\mathbf{M^{k-1}_k}(i,j) = (\phi^{k-1}_i, \phi^k_j)_k = \mathbf{M^k_{k-1}}(j,i). \tag{2.15}$$

2.5. A note on implementing the operators. The fine-grid operator, A_k, the coarse-grid operator, A^G_{k-1}, and the restriction operator, R, are expensive to implement using Equations 2.6, 2.12 and 2.14, respectively. In Appendix E, we show that instead of using these operators, we can solve an equivalent problem using the matrices $\tilde{\mathbf{A}}_\mathbf{k}$, $\tilde{\mathbf{A}}_\mathbf{k-1}$ and $\mathbf{P}^T$ (Equations 2.6 and 2.10). We state the algorithm for the two-level case in Algorithm 1. This scheme can be extended to construct the other standard multigrid schemes, namely the V, W and FMV cycles [16, 17].

3. Implementation. Section 3.1 presents an overview of the octree data structure and its application in finite elements. We discretize the variational problem presented in Section 2 using a sequence of such octree meshes. In Section 3.2, we review the framework introduced in our previous work [50] in which we constructed finite element spaces using conforming, trilinear, basis functions using a 2:1 balanced octree data structure. In Section 3.3, we describe an algorithm for constructing coarse octrees starting with an arbitrary 2:1 balanced fine-grid octree. This sequence of octrees gives rise to a sequence of nested finite element spaces that can be used in the multigrid algorithm presented in Section 2. In Section 3.4, we describe the matrix-free implementation of the restriction and prolongation operators derived in Section 2. Finally, we end this section with a note on variable-coefficient operators.

[6] P is a bounded linear operator from one Hilbert space, V_{k-1}, to another, V_k, and hence it has an unique, bounded, linear Hilbert adjoint operator with respect to the inner-products considered [38].

Algorithm 1. TWO-GRID CORRECTION SCHEME

1. Relax ν_1 times on Equation E.6 with an initial guess, u_k^0. (Pre-smoothing)
2. Compute the fine-grid residual using the solution vector, v_k, at the end of the pre-smoothing step: $\mathbf{r_k} = \tilde{\mathbf{f}}_\mathbf{k} - \tilde{\mathbf{A}}_\mathbf{k}\mathbf{v_k}$.
3. Compute: $\mathbf{r_{k-1}} = \mathbf{P}^T\mathbf{r_k}$. (Restriction)
4. Solve for $\mathbf{e_{k-1}}$ in Equation E.7. (Coarse-grid correction)
5. Correct the fine-grid approximation: $\mathbf{v_k^{new}} = \mathbf{v_k} + \mathbf{P e_{k-1}}$. (Prolongation)
6. Relax ν_2 times on Equation E.6 with the initial guess, v_k^{new}. (Post-smoothing)

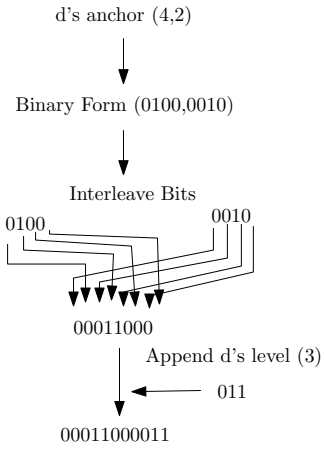


FIG. 3.1. *Computing the Morton id of quadrant "d" in the quadtree shown in Figure 3.2(b). The anchor for any quadrant is it's lower left corner.*

3.1. Review on octrees. An octree[7] is a tree data structure that is used for spatial decomposition (Figure 3.2(b)). Every node[8] of an octree 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 octrees are octrees 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*". An octant's children, grandchildren and so on and so forth are collectively referred to as the octant's "*descendants*" and this octant will be an "*ancestor*" of its descendants. An octant along with all its descendants can be viewed as a separate tree in itself with this octant as its root. Hence, this set is also referred to as a "*subtree*" of the original tree. The depth of an octant from the root is referred to as its "*level*". As shown in Figure 3.2(a), the root of the tree is at level 0 and every interior octant is one level lower than its children.

There are many different ways to represent trees [19]. In this work, we will use a

[7]Sometimes, we use quadtrees for illustration purposes. Quadtrees are 2-D analogues of octrees.
[8]The term "*node*" is usually used to refer to the vertices of elements in a finite element mesh; but, in the context of tree data structures, it refers to the octants themselves.

linearized representation of octrees known as "*linear octrees*". In this representation, we discard the interior octants and only store the complete list of leaves. This is possible only because we use a "*locational code*" to identify the octants. A locational code is a code that contains information about the position and level of the octant in the tree. There are many choices for locational codes [18]; we use the one known as the "*Morton encoding*".

In order to construct a Morton encoding, the maximum permissible depth, $\mathcal{L}_{max}$, of the tree is specified "*a priori*". Note that $\mathcal{L}_{max}$ is different from $\mathcal{L}^*$, the maximum level attained by any octant. In general, $\mathcal{L}^*$ can not be specified a priori. $\mathcal{L}_{max}$ is only an upper bound for $\mathcal{L}^*$. 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 front lower left corner (a_0 in Figure 3.3) of an octant is chosen as its anchor. The Morton encoding for any octant is then derived by interleaving the binary representations of the three coordinates[9] of the octant's anchor, and then appending the binary representation of the octant's level to this sequence of bits [11, 18, 50, 51, 53, 55].

In many applications involving octrees, it is desirable to impose a restriction on the relative sizes of adjacent octants [34, 37, 50, 51, 55]. This is known as the balance constraint. For the applications we're interested in, we enforce a 2:1 "*balance*" constraint:

DEFINITION 1. *A complete, linear d-tree*[10] *with all its octants at levels* $\leq \mathcal{L}^*$ *is 2:1 balanced if and only if, for any* $l \in [1, \mathcal{L}^*)$*, no leaf at level* l *shares an* m*-dimensional face*[11] $(m \in [0, d))$ *with another leaf, at level greater than* $l + 1$.

An example of a 2:1 balanced quadtree is shown in Figure 3.2(c). In the present work, we only work with complete, linear, 2:1 balanced octrees. In this paper, we use the balancing algorithm described in [51].

3.2. Finite elements on octrees. In our previous works [51, 50], we developed low-cost algorithms and efficient data structures for constructing conforming finite element meshes using linear octrees. We use these data structures in the present work too. The key features of this framework are listed below.

- Given a complete linear 2:1 balanced octree, we use the leaves of the octree as the elements of a finite element mesh.
- A characteristic feature of such octree meshes is that they contain "*hanging*" vertices; these are vertices of octants that coincide with the centers of faces or mid-points of edges of other octants. The vertices of the former type are called "*face-hanging*" vertices and those of the latter type are called "*edge-hanging*" vertices. The 2:1 balance constraint ensures that there is at most one hanging vertex on any edge or face.
- We do not store hanging vertices explicitly. They do not represent independent degrees of freedom in a FEM solution. A method to eliminate hanging vertices in locally refined quadrilateral meshes and yet ensure inter-element continuity by the use of special bilinear quadrilateral elements was presented in [57]. We extended that approach to three dimensions. If the i-th vertex of an element/octant is hanging, then the index corresponding to this vertex

[9] We represent the coordinates using integers as shown in Figure 3.2(b).

[10] d-dimensional trees with a maximum of 2^d children per octant are referred to as d-trees.

[11] A corner is a 0-dimensional face, an edge is a 1-dimensional face and a face is a 2-dimensional face.

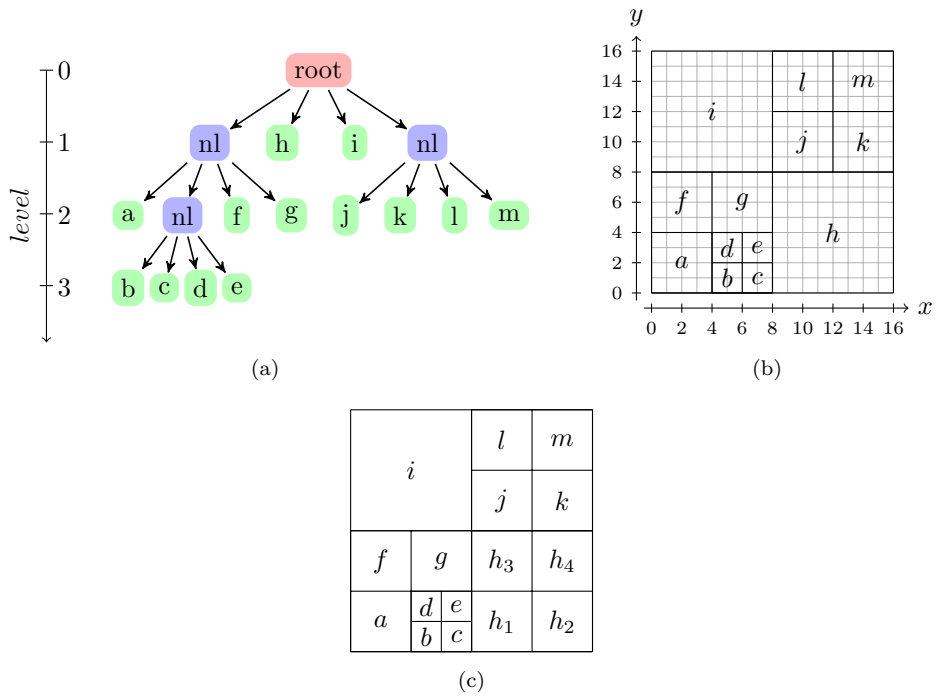


FIG. 3.2. *(a) Tree representation of a quadtree and (b) decomposition of a square domain using the quadtree, superimposed over a uniform grid, and (c) a balanced linear quadtree: result of balancing the quadtree.*

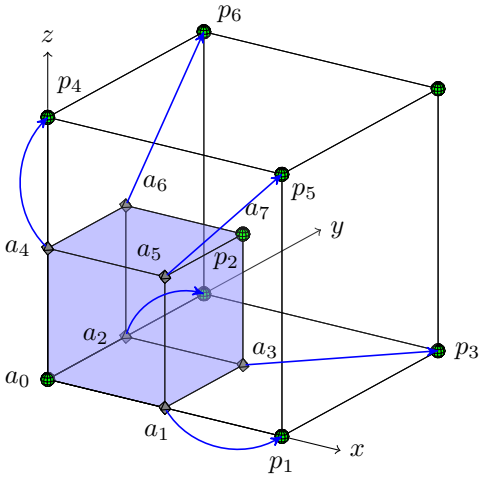


FIG. 3.3. *Illustration of nodal-connectivities required to perform conforming FEM calculations using a single tree traversal. Every octant has at least 2 non-hanging vertices, one of which is shared with the parent and the other is shared amongst all the siblings. The octant shown in blue (a) is a child 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. All the vertices in this illustration are labelled in the Morton ordering.*

will point to the i-th vertex of the parent[12] of this element instead (Figure 3.3). Thus, if a hanging vertex is shared between 2 or more elements, then in each element it might point to a different index.
- Since we eliminate hanging vertices in the meshing stage itself, we don't need to use projection schemes like those used in [3, 35, 37, 56] to enforce conformity. Hence, we don't need multiple tree traversals for performing each MatVec; instead, we perform a single traversal by mapping each octant/element to one of the pre-computed element types, depending on the configuration of hanging vertices for that element.
- To reduce the memory overhead, the linear octree is stored in a compressed form that requires only one byte per octant (the level of the octant). Even the element-to-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.

Below, we list some of the properties of the shape functions defined on octree meshes.
- The shape functions are not rooted at the hanging vertices.
- The shape functions are trilinear.
- The shape functions assume a value of 1 at the vertex at which they are rooted and a value of 0 at all other non-hanging vertices in the octree.
- The support of a shape function can spread over more than 8 elements.
- 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. For example, in Figure 3.3 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. It will assume a value equal to 1 at vertex a_0.
- A shape function assumes non-zero values within an octant if and only if it is rooted at some non-hanging vertex of this octant or if some vertex of the octant under consideration is hanging, say the i-th vertex, and the shape function in question is rooted at the i-th non-hanging vertex of the parent of this octant. Hence, for any octant there are exactly eight shape functions that do not vanish within it and their indices will be stored in the vertices of this octant.
- The finite element matrices constructed using these shape functions are mathematically equivalent to those obtained using projection schemes such as in [37, 55, 56].

To implement finite element MatVecs using these shape functions, we need to enumerate all the permissible hanging configurations for an octant. The following properties of 2:1 balanced linear octrees helps reduce the total number of permissible hanging configurations. Figure 3.3 illustrates these properties.
- Every octant has at least 2 non-hanging vertices and they are:
  - The vertex that is common to both this octant and its parent.
  - The vertex that is common to this octant and all its siblings.
- An octant can have a face hanging vertex only if the remaining vertices on that face are one of the following:
  - Edge hanging vertices.

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

Algorithm 2. FINDING THE CHILD NUMBER OF AN OCTANT

Input: `The anchor (x,y,z) and level (d) of the octant and the maximum permissible depth of the tree (`$\mathcal{L}_{max}$`).`
Output: c, `the child number of the octant.`

1. $l \leftarrow 2^{(\mathcal{L}_{max}-d)}$
2. $l_p \leftarrow 2^{(\mathcal{L}_{max}-d+1)}$
3. $(i,j,k) \leftarrow (x,y,z) \bmod l_p$
4. $(i,j,k) \leftarrow (i,j,k)/l$
5. $c \leftarrow (4k+2j+i)$

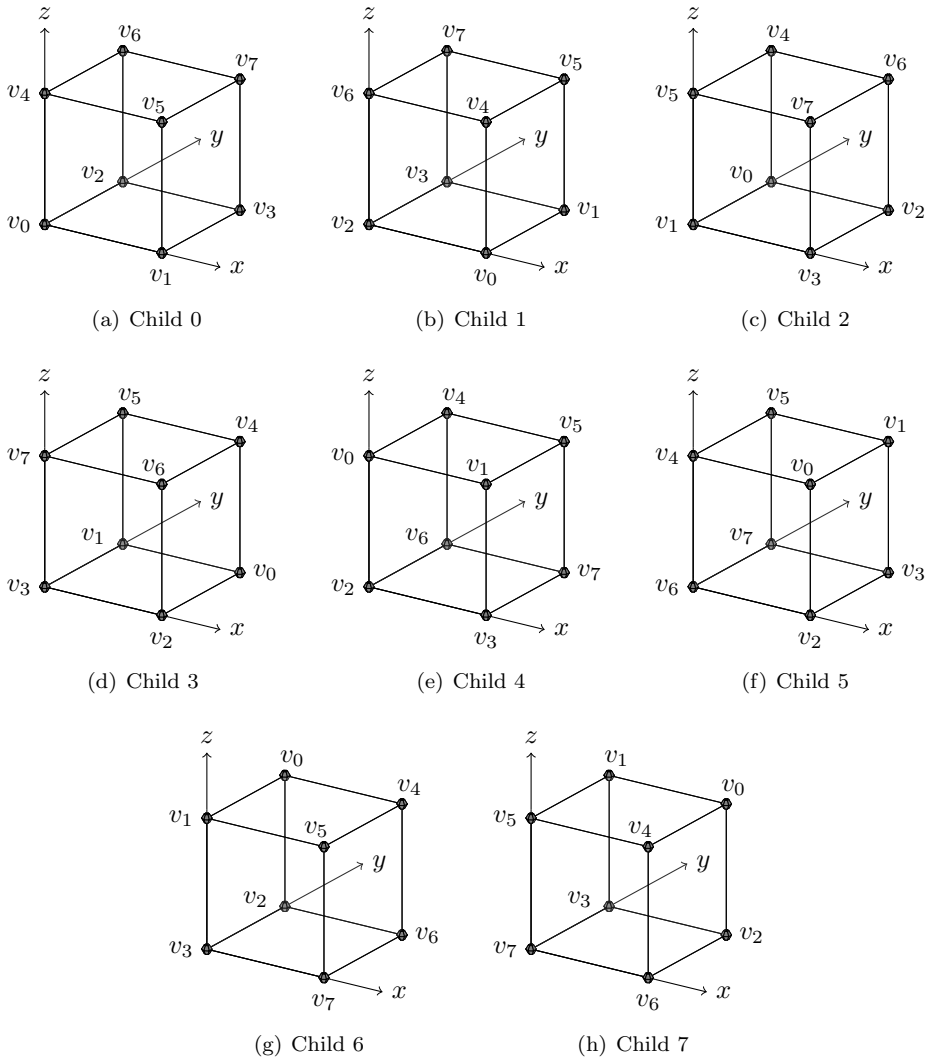
– The vertex that is common to both this octant and its parent.

The "*child number*" of an octant is used to handle hanging vertices in this framework. It is used in the implementation of the FEM MatVecs and the intergrid transfer operations. If an octant shares its k-th vertex with its parent, then it is said to have a child number equal to k. For convenience, we use the Morton ordering to number the vertices of an octant. Thus, sorting the children of an octant in the Morton order is equivalent to sorting the children according to their child numbers. The child number of an octant is a function of the coordinates of its anchor and its level in the tree. Algorithm 2 is used to compute the child number of an octant.

Any element in the mesh belongs to one of the 8 child number based configurations (Figures 3.4(a) - 3.4(h)). In all configurations, v_0 is the vertex that the element shares with its parent and v_7 is the vertex that the element shares with all its siblings. For an element with child number k, v_0 will be the k-th vertex and v_7 will be the $(7-k)$-th vertex. v_0 and v_7 can never be hanging. If v_3, v_5 or v_6 are hanging, they will be face-hanging and not edge-hanging. If v_3 is hanging, then v_1 and v_2 must be edge-hanging. If v_5 is hanging, then v_1 and v_4 must be edge-hanging. If v_6 is hanging, then v_2 and v_4 must be edge-hanging. After factoring in these constraints, there are only 18 potential hanging-vertex configurations for each of the 8 'child number' configurations (Table 3.1).

3.2.1. Overlapping communication with computation. Every octant is owned by a single processor. However, the values of unknowns associated with octants on inter-processor boundaries need to be shared among several processors. We keep multiple copies of the information related to these octants and we term them "*ghost*" octants. In our implementation of the finite element MatVec, each processor iterates over all the octants it owns and also loops over a layer of ghost octants that contribute to the vertices it owns. Within the loops, each octant is mapped to one of the above described hanging configurations (Figures 3.4(a) - 3.4(h)). This is used to select the appropriate element stencil from a list of pre-computed stencils. Although a processor needs to read ghost values from other processors it only needs to write data back to the vertices it owns and does not need to write to ghost vertices. Thus, there is only one communication phase within each MatVec, which we can overlap with a computation phase:

1. Initiate non-blocking MPI sends for information stored on ghost-vertices.
2. Loop over the elements in the interior of the processor domain. These elements do not share any vertices with other processors. We identify these elements during the meshing phase itself.
3. Receive ghost information from other processors.



(a) Child 0 (b) Child 1 (c) Child 2

(d) Child 3 (e) Child 4 (f) Child 5

(g) Child 6 (h) Child 7

FIG. 3.4. *Illustration of the 8 child number configurations. All the octants shown are oriented in the global coordinate system.*

4. Loop over remaining elements to update information.

3.3. 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 is either present in the $k+1$-th octree as well or all its eight children are present instead (Figures 3.5(a) - 3.5(c)).

We construct the k-th octree from the $k+1$-th octree by replacing every set of eight siblings by their parent. This algorithm is based on the fact that in a sorted linear octree, each of the 7 successive elements following a "Child-0" element is either one of its siblings or a decendant of its siblings. Let i and j be the indices of any two successive Child-0 elements in the $k+1$-th octree. We have the following 3 cases: (a) $j < (i+8)$, (b) $j = (i+8)$ and (c) $j > (i+8)$. In the first case, the elements with

Element Type	Corresponding Hanging vertices
1	None
2	v_2
3	v_1
4	v_1 and v_2
5	v_4
6	v_4 and v_2
7	v_4 and v_1
8	v_4, v_1 and v_2
9	v_3, v_1 and v_2
10	v_3, v_4, v_1 and v_2
11	v_6, v_4 and v_2
12	v_6, v_4, v_1 and v_2
13	v_6, v_3, v_4, v_1 and v_2
14	v_5, v_4 and v_1
15	v_5, v_4, v_1 and v_2
16	v_5, v_3, v_4, v_1 and v_2
17	v_5, v_6, v_4, v_1 and v_2
18	v_5, v_6, v_3, v_4, v_1 and v_2

TABLE 3.1
The list of permissible hanging vertex configurations for any octant.

indices in the range $[i, j)$ are not coarsened. In the second case, the elements with indices in the range $[i, j)$ are all siblings of each other and are replaced by their parent. In the last case, the elements with indices in the range $[i, (i + 7)]$ are all siblings of each other and are replaced by their parent. The elements with indices in the range $[(i + 8), j)$ are not coarsened. The pseudocode for the sequential implementation of the coarsening algorithm is given in Algorithm 3.

Coarsening is an operation with $\mathcal{O}(N)$ work complexity, where N is the number

Algorithm 3. Sequential Coarsening

Input: A sorted, complete, linear fine octree (F).
Output: A sorted, complete linear coarse octree (C).
Note: This algorithm can also be used with a contiguous subset of F, provided the first element of this subset is a Child-0 element and the last element of this subset is either the last element of F or the element that immediately precedes a Child-0 element. The output in this case will be the corresponding contiguous subset of C.

```
1.  C ← ∅
2.  I1 ← 0
3.  while (I1 < len(F))
4.        Find I2 such that Child − Number(F[I2]) = 0 and
          Child − Number(F[k]) ≠ 0  ∀  I1 < k < I2.
5.        if no such I2 exists
6.           I2 ← len(F)
7.        end if
8.        if I2 ≥ (I1 + 8)
9.           C.push_back(Parent(F[I1]))
10.          if I2 > (I1 + 8)
11.             C.push_back(F[I1 + 8], F[I1 + 9], ..., F[I2 − 1])
12.          end if
13.       else
14.          C.push_back(F[I1], F[I1 + 1], ..., F[I2 − 1])
15.       end if
16.       I1 ← I2
17. end while
```

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 processors.[13] The main parallel operations are two circular shifts; one clockwise and another anti-clockwise. The message in each case is just 1 integer: (a) the index of the first Child-0 element on each processor and (b) the number of elements between the last Child-0 element on any processor and the last element on that processor. While we communicate these messages in the background, we simultaneously process the elements in between the first and last Child-0 elements on each processor. The pseudocode for the parallel implementation of the coarsening algorithm is given in Algorithm 4.

However, the operation described above may produce 4:1 balanced octrees[14] instead of 2:1 balanced octrees. Hence, we balance the result using the algorithm described in [51]. This balancing algorithm has an $\mathcal{O}(N \log N)$ work complexity and $\mathcal{O}(\frac{N}{n_p} \log \frac{N}{n_p} + n_p \log n_p)$ parallel time complexity. 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 processors.

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

[14] 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.

Algorithm 4. PARALLEL COARSENING
(AS EXECUTED BY PROCESSOR P)

Input: A distributed, globally sorted, complete, linear fine octree (F).
Output: A distributed, globally sorted, complete, linear coarse octree (C).
Note: We assume that $\mathbf{len}(F) > 8$ on each processor.

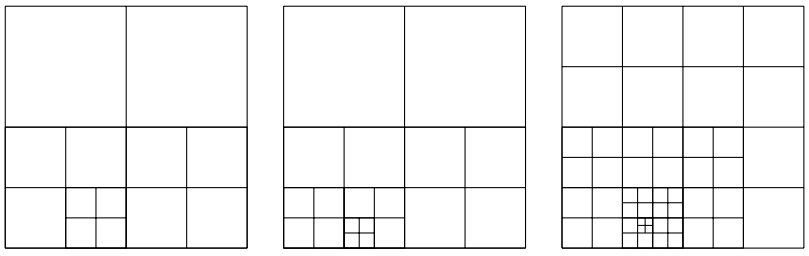
1. $C \leftarrow \emptyset$
2. Find $\mathcal{I}_f$ such that $\mathbf{Child-Number}(F[\mathcal{I}_f]) = 0$ and $\mathbf{Child-Number}(F[k]) \neq 0 \quad \forall \quad 0 \leq k < \mathcal{I}_f$.
3. **if** no such $\mathcal{I}_f$ exists on P
4. $M_f \leftarrow -1$; $M_l \leftarrow -1$
5. **else**
6. Find $\mathcal{I}_l$ such that $\mathbf{Child-Number}(F[\mathcal{I}_l]) = 0$ and $\mathbf{Child-Number}(F[k]) \neq 0 \quad \forall \quad \mathcal{I}_l < k < \mathbf{len}(F)$.
7. $M_f \leftarrow \mathcal{I}_f$; $M_l \leftarrow (\mathbf{len}(F) - \mathcal{I}_l)$
8. **end if**
9. **if** P is not the first processor
10. Send M_f to the previous processor (P-1) using an non-blocking MPI send.
11. **end if**
12. **if** P is not the last processor
13. Send M_l to the next processor (P+1) using an non-blocking MPI send.
14. **else if** $M_f > -1$
15. $\mathcal{I}_l \leftarrow \mathbf{len}(F)$
16. **end if**
17. **if** $M_f > -1$
18. Coarsen the list $\{F[\mathcal{I}_f], F[\mathcal{I}_f + 1], \ldots, F[\mathcal{I}_l - 1]\}$ and store the result in C. (**Algorithm 3**)
19. **end if**
20. **if** P is not the first processor
21. Receive $\mathcal{I}_p$ from the previous processor (P-1).
22. Process octants with indices $< \mathcal{I}_f$. (**Algorithm 5**)
23. **end if**
24. **if** P is not the last processor
25. Receive $\mathcal{I}_n$ from the next processor (P+1).
26. Process octants with indices $\geq \mathcal{I}_l$. (**Algorithm 6**)
27. **end if**

The sequence of octrees constructed as described above has the property that non-hanging vertices in any octree remain non-hanging in all the 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.

3.4. Intergrid transfer operations. To implement the intergrid transfer operations in Algorithm 1, we need to find all the non-hanging fine-grid vertices that

Algorithm 5. COARSENING THE FIRST FEW OCTANTS ON PROCESSOR P (SUBCOMPONENT OF ALGORITHM 4)

```
1.  if I_p ≥ 0 and M_f ≥ 0
2.    if (I_p + I_f) ≥ 8
3.      I_c ← max(0, (8 − I_p))
4.      C.push_front(F[I_c], F[I_c + 1], ..., F[I_f − 1])
5.    else
6.      C.push_front(F[0], F[1], ..., F[I_f − 1])
7.    end if
8.  else
9.    if M_f < 0
10.     if I_p < 0 or I_p ≥ 8
11.       C ← F
12.     else
13.       I_c ← (8 − I_p)
14.       C.push_front(F[I_c], F[I_c + 1], ..., F[I_f − 1])
15.     end if
16.   else
17.     C.push_front(F[0], F[1], ..., F[I_f − 1])
18.   end if
19. end if
```



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

FIG. 3.5. *Quadtree meshes for three succesive multigrid levels.*

lie within the support of each coarse-grid shape function. This is trivial on regular grids. However, for unstructured grids this can be quite expensive; especially for parallel implementations. Fortunately, for a hierarchy of octree meshes constructed as described in Section 3.3, these operations can be implemented quite efficiently.

As seen in Section 2.5, the restriction matrix is the transpose of the prolongation matrix. We do not construct these matrices explicity, instead we implement a matrix-free scheme using MatVecs. The MatVecs for the restriction and prolongation operators are very similar. In both cases, we loop over the coarse and fine grid octants simultaneously. For each coarse-grid octant, the underlying fine-grid octant could either be the same as itself or be one of its eight children (Section 3.3). 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 (Section 3.2) and evaluating them at the non-hanging fine-grid vertices that lie within

Algorithm 6. COARSENING THE LAST FEW OCTANTS ON PROCESSOR P (SUBCOMPONENT OF ALGORITHM 4)

```
1.  if I_n ≥ 0 and M_l ≥ 0
2.    if (I_n + M_l) ≥ 8
3.      C.push_back(Parent(F[I_l]))
4.      if M_l > 8
5.        C.push_back(F[I_l + 8], F[I_l + 9], ..., F[len(F) − 1])
6.      end if
7.    else
8.      C.push_back(F[I_l], F[I_l + 1], ..., F[len(F) − 1])
9.    end if
10. else
11.   if M_l ≥ 0
12.     C.push_back(Parent(F[I_l]))
13.     if M_l > 8
14.       C.push_back(F[I_l + 8], F[I_l + 9], ..., F[len(F) − 1])
15.     end if
16.   end if
17. end if
```

this coarse-grid octant. These form the entries of the restriction and prolongation matrices (Equation 2.10).

To parallelize this operation, we need the coarse and fine grid partitions to be "*aligned*". By aligned we require the following two conditions to be satisfied:

- If an octant exists both in the coarse and fine grids, then the same processor must "*own*" this octant on both the meshes.
- If an octant's children exist in the fine-grid, then the same processor must own this octant on the coarse mesh and all its 8 children on the fine mesh.

In order 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 the multigrid levels. For example, the coarser levels might be too sparse to be distributed across all the processors or using the same partition for all the multigrid levels could lead to a large load imbalance across the processors. Hence, we allow some multigrid levels to be partitioned differently than others.[15] When a transition in the partitions is required, we duplicate the octree in question and let one of the duplicates share the same partition as that of its immediate finer level and let the other one share the same partition as that of its immediate coarser level. We refer to one of these duplicates as the "*pseudo*" mesh (Figure 3.6(b)). The pseudo mesh is only used to support intergrid transfer operations (Smoothing is not performed on this mesh). On these multigrid 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 keep track of the pairs of coarse and fine grid vertices that were visited already. In order to implement this MatVec efficiently, we

[15] It is also possible that some processors are idle on the coarse-grids, while no processor is idle on the finer grids.

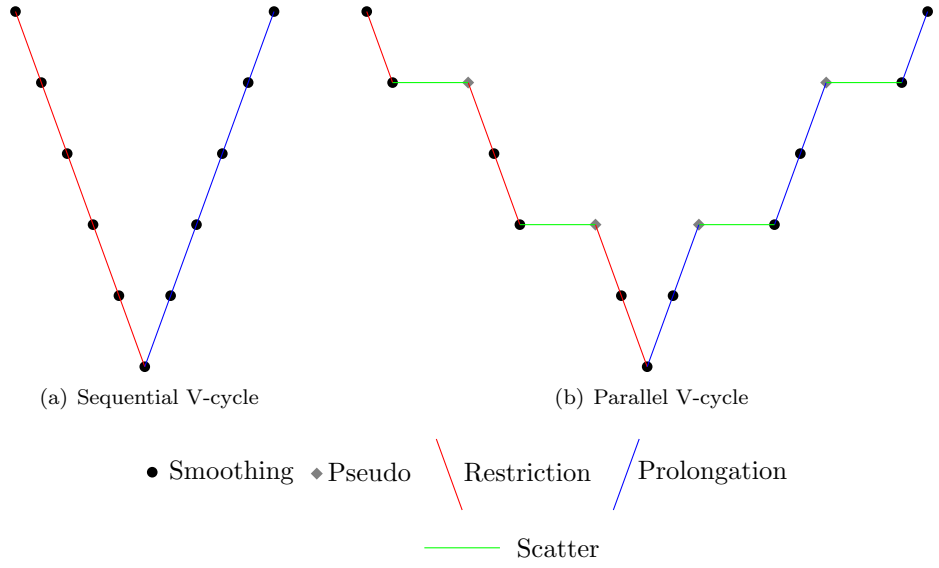


FIG. 3.6. *(a) A V-cycle where the meshes at all multigrid levels share the same partition and (b) A V-cycle where not all meshes share the same partition. Some meshes do share the same partition and whenever the partition changes a pseudo mesh is added. The pseudo mesh is only used to support intergrid transfer operations and smoothing is not performed on this mesh.*

make use of the following observations.

- Every non-hanging fine-grid vertex is shared by at most eight fine-grid elements, excluding the elements whose hanging vertices are mapped to this vertex.
- Each of these eight fine-grid elements will be visited only once within the Restriction and Prolongation MatVecs.
- Since we loop over the coarse and fine elements simultaneously, there is a coarse octant associated with each of these eight fine octants. These coarse octants (maximum of eight) overlap with the respective fine octants.
- 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. So, the correct index is always stored immaterial of the hanging state of the vertex.

We pre-compute 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 to determine whether or not the respective coarse-grid shape function contributes to this fine-grid vertex. The overhead of using this mask within the actual MatVecs is just the cost of a few bitwise operations for each fine-grid octant. Algorithm 7 lists the sequence of operations performed by a processor 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 communica-

Algorithm 7. Parallel Restriction MatVec
(As executed by processor P)

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 processors.
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 processors.
28. Add the contributions recieved from other processors
    to the local copy of C.
```

tion with computation in the pseudocode. In the actual implementation, we overlap communication with computation as described in Section 3.4.2. The following section describes how we compute these masks for any given pair of coarse and fine octrees.

3.4.1. 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 contributes to this fine vertex could be rooted.

[16]This is a weak upper bound.

Each of the vertices of the coarse-grid octants that 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.

Now, we briefly describe how we identified these 1758 cases. We first choose one of the eight fine-grid octants surrounding a given fine-grid vertex as a reference element. Without loss of generality, we pick the octant whose anchor is located at the given fine vertex. Now the remaining fine-grid octants could either be the same size as the reference element, or be half the size or twice the size of the reference element. This simply follows from the 2:1 balance constraint. Further, each of these eight fine-grid octants could either be the same as the overlapping coarse-grid octant or be any of its eight children. Moreover, each of these coarse-grid octants that overlap the fine-grid octants under consideration could belong to any of the 8 child number types, each of which could further be of any of the 18 hanging configurations. Taking all these possible combinations into account, we can locate all the possible non-hanging coarse-grid vertices around a fine-grid vertex. Note that the child numbers, the hanging vertex configurations, and relative sizes of the eight fine-grid octants described above are not mutually independent. Each choice of child number, hanging vertex configuration and size for one of the eight fine-grid octants imposes numerous constraints on the respective choices for the other elements. However, to list all these possible constraints would be a complicated excercise and it is unnecessary for our purposes. Instead, we simply assume that the choices for the eight elements under consideration are mutually independent. This computation can be done offline and results in a weak upper bound of 1758 unique non-hanging coarse-grid locations around any fine-grid vertex.

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 "*PreMatVec*" before we actually begin solving the problem; this is only performed once for each multigrid level. In this PreMatVec, 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 for each of the possibilites described above. These flags contain the following information.

- A flag to determine whether or not the coarse and fine grid octants are the same (1 bit).
- The child number of the current fine-grid octant (3 bits).
- The child number of the corresponding coarse-grid octant (3 bits).
- The hanging configuration of the corresponding coarse-grid octant (5 bits).
- 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 PreMatVec is an operation with $\mathcal{O}(N)$ work complexity and has an $\mathcal{O}(\frac{N}{n_p})$ parallel time complexity.

3.4.2. Overlapping communication with computation. Finally, we overlap computation with communication for ghost values even within the Restriction and Prolongation MatVecs. However, unlike the finite element MatVec the loop is split into three parts because we can not loop over ghost octants since these octants need not be aligned across grids. Hence, each processor loops only over the coarse and the

underlying fine octants that it owns. As a result, we need to both read as well as write to ghost values within the MatVec. The steps involved are listed below:

1. Initiate non-blocking MPI sends for ghost-values from the input vector.
2. Loop over some of the coarse and fine grid elements that are present in the interior of the processor domains. These elements do not share any vertices with other processors.
3. Recieve the ghost-values sent from other processors in step 1.
4. Loop over the coarse and fine grid elements that share at least one of its vertices with a different processor.
5. Initiate non-blocking MPI sends for ghost-values in the output vector.
6. Loop over the remaining coarse and fine grid elements that are present in the interior of the processor domains. Note in step 2, we only iterated over some of these elements. In this step, we iterate over the remaining elements.
7. Recieve the ghost-values sent from other processors in step 5.
8. Add the values recieved in step 7 to the existing values in the output vector.

3.5. Handling variable-coefficient operators. One of the problems with geometric multigrid methods is that their performance deteriorates with increasing contrast in material properties [17, 21]. Section 2.5 shows that the direct coarse-grid discretization can be used instead of the Galerkin coarse-grid operator 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. For variable-coefficient problems, this means that 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. Hence, the convergence of the stand-alone multigrid solver deteriorates with increasing contrast in material properties. The standard solution is to use multigrid as a preconditioner to the Conjugate Gradient (CG) method [52]. We have conducted numerical experiments that demonstrate this for the Poisson problem. The method works well for smooth coefficients but it is not robust in the presence of discontinuous coefficients.

3.6. Summary. The sequence of steps involved in solving the problem defined in Section 2.1.1 is summarized below:

1. A "sufficiently" fine[17] 2:1 balanced complete linear octree is constructed using the algorithms described in [51].
2. Starting with the finest octree, a sequence of 2:1 balanced coarse linear octrees is constructed using the global coarsening algorithm (Section 3.3).
3. The maximum number of processors that can be used for each multigrid level without violating the minimum grain size criteria (Appendix F) is computed.
4. Starting with the coarsest octree, the octree at each multigrid level is meshed using the algorithm described in [50]. As long as the load imbalance across the processors is acceptable and as long as the number of processors used for the coarser grid is the same as the maximum number of processors that can be used for the finer level without violating the minimum grain size criteria, the partition of the coarser grid is imposed on to the finer grid during meshing.

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

Algorithmic Component	Parallel Time Complexity
Octree Construction	$\mathcal{O}(\frac{N}{n_p}\log(\frac{N}{n_p}) + n_p \log n_p)$
2:1 Balancing	$\mathcal{O}(\frac{N}{n_p}\log(\frac{N}{n_p}) + n_p \log n_p)$
Partition	$\mathcal{O}(\frac{N}{n_p} + n_p)$
Meshing	$\mathcal{O}(\frac{N}{n_p}\log(\frac{N}{n_p}) + n_p \log n_p)$
Coarsening (without balancing) 1 Level	$\mathcal{O}(\frac{N}{n_p})$
Matvecs	$\mathcal{O}(\frac{N}{n_p})$

TABLE 3.2

Complexity estimates for the various algorithmic components assuming a Hypercube network topology with $\theta(n_p)$ bandwidth. N is the size of the linear octree and n_p is the number of processors.

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. The process is repeated until the finest octree has been meshed.

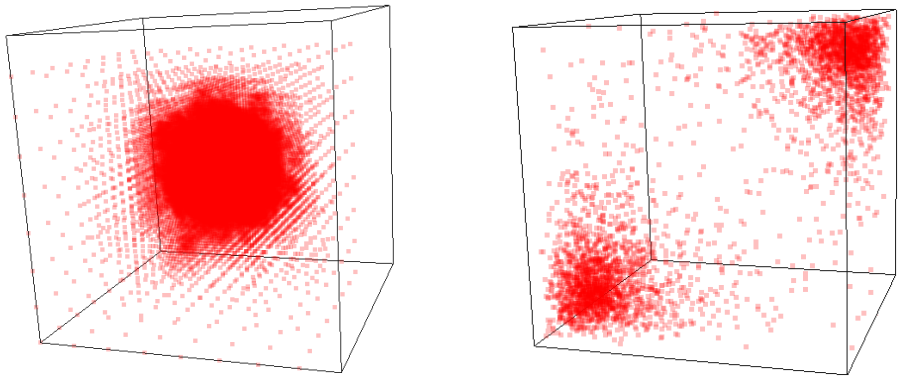
5. A restriction PreMatVec (Section 3.4) is performed at each multigrid level (except the coarsest) and the masks that will be used in the actual restriction and prolongation MatVecs are computed and stored.
6. For the case of variable-coefficient operators, vectors that store the material properties at each multigrid level are created.
7. The discrete system of equations is then solved using the conjugate gradient algorithm preconditioned with the multigrid scheme.

Table 3.2 gives the parallel time complexity of the various algorithmic components as a function of the problem size, N, and the number of processors, n_p.

4. Numerical experiments. We consider the following 3-dimensional, scalar, linear elliptic problems with homogeneous Neumann boundary conditions:

$$\begin{aligned} -\Delta u + u &= f \text{ in } \Omega \\ \hat{n}\cdot\nabla u &= 0 \text{ in } \partial\Omega \\ \Omega &= [0,1]\times[0,1]\times[0,1] \end{aligned} \tag{4.1}$$

$$\begin{aligned} -\nabla\cdot(\epsilon\nabla u) + u &= f \text{ in } \Omega \\ \hat{n}\cdot\nabla u &= 0 \text{ in } \partial\Omega \\ \Omega &= [0,1]\times[0,1]\times[0,1] \\ \epsilon(x,y,z) &= \left(1 + 10^6\left(\cos^2(2\pi x) + \cos^2(2\pi y) + \cos^2(2\pi z)\right)\right) \end{aligned} \tag{4.2}$$



(a) (b)

FIG. 4.1. *Samples of the point distributions used for the numerical experiments: (a) A Guassian point distribution with mean at the center of the unit cube and (b) A log-normal point distribution with mean near one corner of the unit cube and it's mirror image about the main diagonal.*

$$
\begin{aligned}
-\nabla \cdot (\epsilon \nabla u) + u &= f \text{ in } \Omega \\
\hat{n} \cdot \nabla u &= 0 \text{ in } \partial\Omega \\
\Omega &= [0,1] \times [0,1] \times [0,1] \\
\epsilon(x,y,z) &= \begin{cases} 10^7 & \text{if } 0.3 \le x,y,z \le 0.6 \\ 1.0 & \text{otherwise} \end{cases}
\end{aligned} \tag{4.3}
$$

We discretized these problems on various octree meshes generated using Gaussian and log-normal distributions.[18] Figures 4.1(a) and 4.1(b) respectively show samples of the Gaussian and log-normal distributions that were used in all our experiments. The number of elements in these meshes range from about 25 thousand to over 1 billion and were solved on up to 4096 processors on Pittsburgh Supercomputing center's Cray XT3 MPP system Bigben and Teragrid's NCSA Intel 64 Linux Cluster Abe. Bigben is equipped with 2068 compute nodes; each node has two 2.6 GHz AMD Opteron processors (1 MB cache for each CPU) and the two processors on a node share 2 GB RAM. The peak performance is approximately 20 Tflops. The nodes are connected using a custom-designed interconnect. Abe comprises of 1200 nodes with a total of 9600 CPUs. Each node is equipped with an Intel 64 (Clovertown) 2.33 GHz dual socket quad core processor, which has 2 MB L2 cache per core and 8 GB/16 GB RAM per node. The peak performance is approximately 89.47 Tflops. The nodes are connected by an InfiniBand interconnect. Our C++ implementation uses MPI, PETSc [7] and SuperLU_Dist [39]. The runs were profiled using PETSc.

In this section, we present the results from 4 sets of experiments: `(A)` A convergence test, `(B)` Isogranular scalability, `(C)` Fixed size scalability and `(D)` Comparison with an off-the-shelf algebraic multigrid implementation. The parameters used in the experiments are listed below:

[18] In the following experiments, the octrees were not generated based on the underlying material properties. In [47], we give some examples for constructing octrees based on user-supplied data such as material properties and source terms.

Maximum Element Size (h_{max})	L^2 norm of the error
$\frac{1}{16}$	3.98×10^{-3}
$\frac{1}{32}$	9.62×10^{-4}
$\frac{1}{64}$	2.46×10^{-4}
$\frac{1}{128}$	6.18×10^{-5}
$\frac{1}{256}$	1.56×10^{-5}

TABLE 4.1

L^2 norm of the error between the true solution and its finite element approximation for the variable coefficient problem (Equation 4.2). The sequence of meshes used in this experiment were constructed by using a base discretization of $\approx$ 0.25M elements generated using a gaussian point distribution followed by successive uniform refinements of the coarse elements of this mesh.

- For experiment `(A)`, we set $u = \cos(2\pi x)\cos(2\pi y)\cos(2\pi z)$ and constructed the corresponding force (f).
- For experiments `(B)`, `(C)` and `(D)`, we used a random solution (u) to construct the force (f).
- A zero initial guess was used in all experiments.
- One multigrid V-cycle was used as a preconditioner to the Conjugate Gradient (CG) method in all experiments. This is known to be more robust than the stand alone multigrid algorithm for variable-coefficient problems [52].
- The damped Jacobi method was used as the smoother at each multigrid level.
- SuperLU_Dist was used to solve the coarsest grid problem in all cases.
- In order to minimize communication costs, the coarsest multigrid level used fewer processors than the finer levels. This keeps the setup cost for SuperLU_Dist low.

4.1. Convergence Test. In the first experiment, a base discretization of approximately $\approx$ 0.25M elements generated using the Gaussian distribution was used to solve the variable-coefficient problem (Equation 4.2). We measured the L^2 norm of the error as a function of the maximum element size (h_{max}) by uniformly refining the coarse elements[19] in this base mesh. In Table 4.1, we report the L^2 norm of the error between the true solution and its finite element approximation for the sequence of meshes constructed as described above. A second order convergence is observed just as predicted by the theory.

4.2. Scalability Results. The fixed size and iso-granular scalability experiments were performed on Bigben. In all the fixed-size and iso-granular 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, 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

[19] Any element whose length is greater than h_{max}.

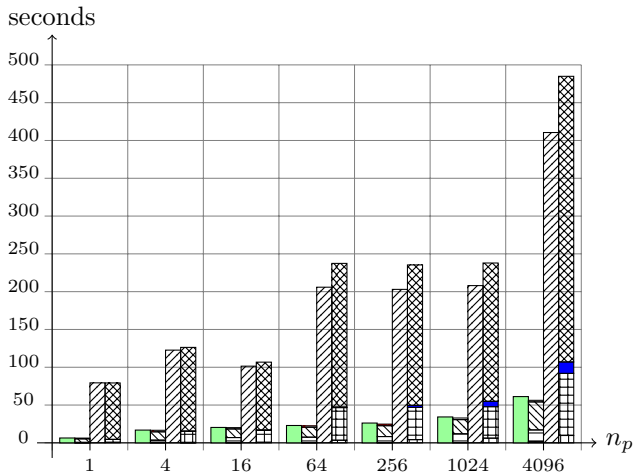
the sum of the individual components.

4.2.1. Isogranular (Weak) scalability. Isogranular scalability analysis was performed by tracking the execution time while increasing the problem size and the number of processors proportionately. The results from isogranular scalability experiments on the octrees generated from Gaussian point distributions are reported in Figures 4.2 and 4.3. Figure 4.2 reports the results for the constant coefficient (Equation 4.1) problem and Figure 4.3 reports the results for the variable-coefficient problem (Equation 4.2). The results from an isogranular scalability experiment for solving the variable-coefficient problem (Equation 4.2) on octrees generated from log-normal point distributions are reported in Figure 4.4. There is little variation between the Gaussian distribution case and the log-normal distribution case. It is quite promising that the setup costs are smaller than the solution costs, suggesting that the method 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. Load balancing is a challenging problem due to the following reasons:

- 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.
- 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.
- Partitioning each multigrid level independently to get good load balance for the smoothing operations at each multigrid level would require the creation of an auxillary mesh for each multigrid level and a scatter operation for each intergrid transfer operation at each multigrid level. This would increase the setup costs and the communication costs.

4.2.2. Fixed-size (Strong) scalability. Fixed-size scalability was performed on the octrees generated from Gaussian and log-normal point distributions to compute the speedup when the problem size is kept constant and the number of processors is increased. The results from fixed size scalability experiments for the constant coefficient problem (Equation 4.1) solved on an octree with 32M (approx) elements generated from Gaussian point distribution are reported in Figure 4.5. The corresponding results for solving the variable-coefficient problem (Equation 4.2) on the same octree are reported in Figure 4.6. This experiment was repeated on octrees with 6M and 22M (approx) elements generated from log-normal point distributions and the corresponding results are reported in Figures 4.8 and 4.7, respectively. The results for the Gaussian and log-normal distributions are similar. We observe nearly ideal speed-ups for the setup phase on up to 256 processors and the speed-ups begin to deteriorate beyond that. We believe that the surface computation (e.g. meshing for ghost elements) begins to dominate beyond 256 processors. Note that the number of meshes also grow with the number of processors. This is another reason why we don't observe ideal speed-ups for the setup phase. The speed-ups for the solve phase, although not ideal, seem to be quite good. Poor load balancing, which affects isogranular scalability on large processor counts, could be another factor that affects the speed-ups for the setup and solve phases in the fixed-size scalability experiments.

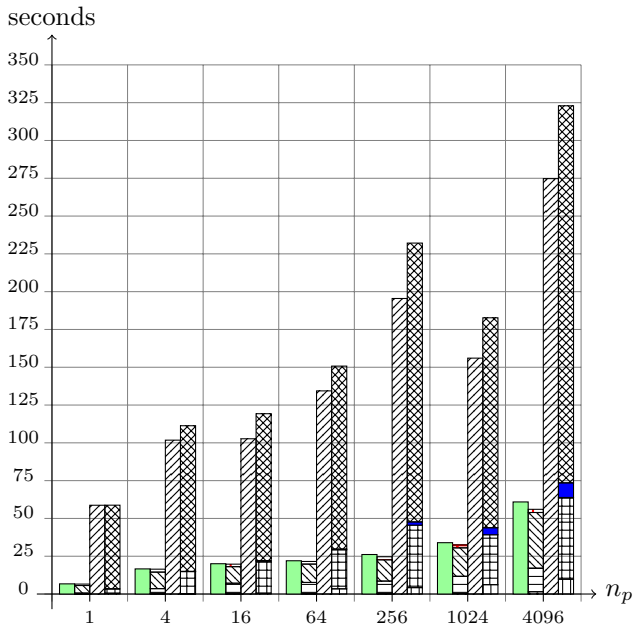
4.3. Comparison with BoomerAMG. Finally, the results from the comparison between the geometric multigrid and algebraic multigrid (BoomerAMG) schemes



	1	4	16	64	256	1024	4096
Coarsening	0.047	0.148	0.146	0.149	0.183	0.288	1.5
Balancing	0.58	3.41	6.86	7.36	8.14	11.49	15.6
Meshing	4.67	11.05	11.4	13.16	14.39	18.97	37.1
R-setup	0.982	2.01	1.74	1.84	2.07	2.12	2.08
Total Setup	**6.42**	**16.88**	**20.4**	**22.97**	**26.21**	**34.28**	**61.23**
LU	0.545	0.115	0.57	3.45	4.25	6.19	9.72
R + P	3.998	15.36	15.99	43.19	42.69	41.67	82.3
Scatter	0	0	0.268	0.765	2.64	7.13	15.04
FE Matvecs	74.73	110.88	89.99	190.13	185.92	183.06	378.1
Total Solve	**79.396**	**122.73**	**101.5**	**206.02**	**203.14**	**208.13**	**410.76**
CG Its.	7	7	7	8	8	8	8
Meshes	4	7	8	9	11	14	16
MG Levels	4	7	7	7	8	9	10
Finest Octant's Level	8	14	14	16	18	19	21

FIG. 4.2. *Isogranular scalability for solving constant coefficient Poisson problems (Equation 4.1) on a set of octrees with a grain size of 0.25M elements (approx.) per processor (n_p) generated using a Gaussian distribution of points. 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. The iterations were terminated when the 2-norm of the residual was reduced by a factor of 10^{-14}. The number of outer CG iterations required are reported. The number of multigrid levels used in each case is also reported. For the single processor case, only 4 multigrid levels were used because the grain size on the coarsest grid would have fallen below 1000 elements (the minimum grain size) had we used more multigrid levels. The total number of meshes generated for each case is also reported. Note that due to the addition of auxillary meshes, the total number of meshes is greater than the number of multigrid levels.*

for the variable-coefficient problem (Equation 4.3) are reported in Figure 4.9. This experiment was performed on Abe. For BoomerAMG, we experimented with two different coarsening schemes: Falgout coarsening and CLJP coarsening. The results from both experiments are reported. [33] reports that Falgout coarsening works best for structured grids and CLJP coarsening is better suited for unstructured grids. Since octree meshes lie in between both structured and generic unstructured grids, we com-

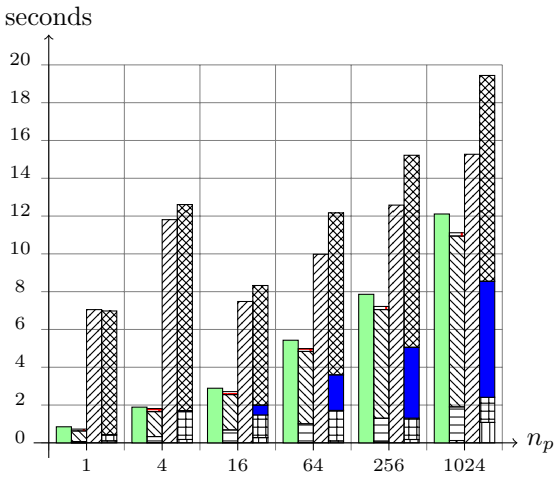


	1	4	16	64	256	1024	4096
Coarsening	0.047	0.145	0.146	0.147	0.182	0.288	1.51
Balancing	0.57	3.41	7.0	7.51	8.38	11.44	15.58
Meshing	5.01	10.93	10.95	12.13	14.06	18.8	36.82
R-setup	0.946	1.95	1.66	1.75	2.04	1.96	2.11
Total Setup	**6.72**	**16.62**	**20.04**	**22.0**	**26.13**	**33.95**	**60.9**
LU	0.54	0.116	0.566	3.43	4.23	6.18	9.8
R + P	2.86	14.72	21.21	25.81	41.31	33.02	54.07
Scatter	0	0.124	0.228	0.49	2.15	4.64	9.64
FE Matvecs	55.36	96.33	97.24	121.0	184.4	138.84	249.36
Total Solve	**58.71**	**101.76**	**102.67**	**134.34**	**195.45**	**156.0**	**274.71**
Elements	239.4K	995.4K	3.97M	16.0M	64.4M	256.8M	1.04B
Vertices	151.7K	660.1K	2.68M	10.52M	42.0M	172.4M	702.9M

FIG. 4.3. *Isogranular scalability for solving the variable-coefficient Poisson problem (Equation 4.2) on the set of octrees used in Figure 4.2. The iterations were terminated when the 2-norm of the residual was reduced by a factor of* 10^{-10}. *5 iterations were required in each case. The number of multigrid levels and meshes used in each case are the same as in Figure 4.2.*

pare our results using both the schemes. While the convergence rate of the geometric multigrid method deteriorates with increasing contrast in the material properties, the convergence rate of BoomerAMG is not affected by contrasts in the material properties. Hence, the variable-coefficient problem is more interesting for this experiment. This experiment shows that GMG has a much lower setup cost compared to AMG; the solution costs are comparable. The AMG scheme seems to have a higher convergence rate since it takes fewer iterations as compared to GMG.

5. Conclusions. We have described a parallel geometric multigrid method for solving elliptic partial differential equations using finite elements on octree based

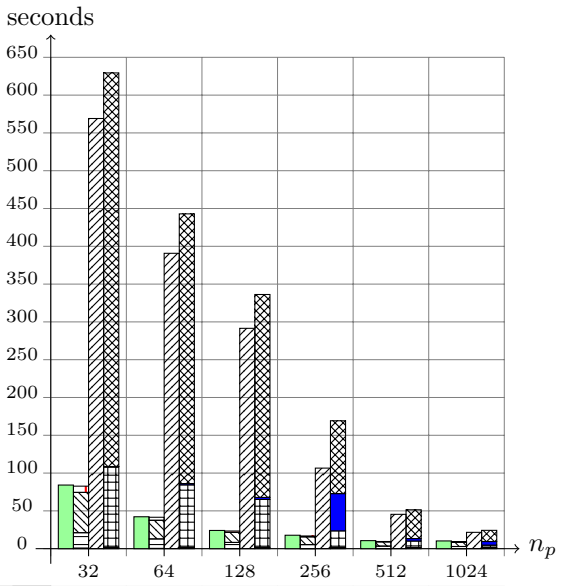


	1	4	16	64	256	1024
Coarsening	0.005	0.013	0.013	0.016	0.042	0.123
Balancing	0.059	0.319	0.67	0.99	1.26	1.78
Meshing	0.565	1.32	1.87	3.83	5.75	9.04
R-setup	0.098	0.162	0.158	0.151	0.165	0.174
Total Setup	**0.85**	**1.89**	**2.89**	**5.43**	**7.86**	**12.11**
LU	0.102	0.178	0.276	0.098	0.177	1.08
R + P	0.315	1.47	1.2	1.62	1.13	1.33
Scatter	0	0.061	0.52	1.88	3.75	6.14
FE Matvecs	6.56	10.9	6.33	8.57	10.16	10.89
Total Solve	**7.05**	**11.81**	**7.48**	**9.98**	**12.58**	**15.27**
CG Its.	5	5	5	6	5	5
Meshes	3	5	8	13	15	15
MG Levels	3	4	5	7	8	8
Finest Octant's Level	9	13	13	13	15	16
Coarsest Octant's Level	3	3	4	4	5	5
Elements	24.6K	99.3K	362.6K	1.42M	5.64M	22.4M
Vertices	17.4K	68.2K	243.3K	952.2K	3.79M	14.9M

FIG. 4.4. *Isogranular scalability for solving the variable-coefficient Poisson problem (Equation 4.2) on a set of octrees with a grain size of 25K elements (approx.) per processor (n_p) generated using a log-normal distributions of points located on two diagonally opposite corners of the unit cube. The iterations were terminated when the 2-norm of the residual was reduced by a factor of 10^{-10}. The levels of the coarsest and finest octants at the finest multigrid level are reported in the figure.*

discretizations. The features of the described method are summarized below:

- 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 to satisfy any user-defined constraint such as a limit on the

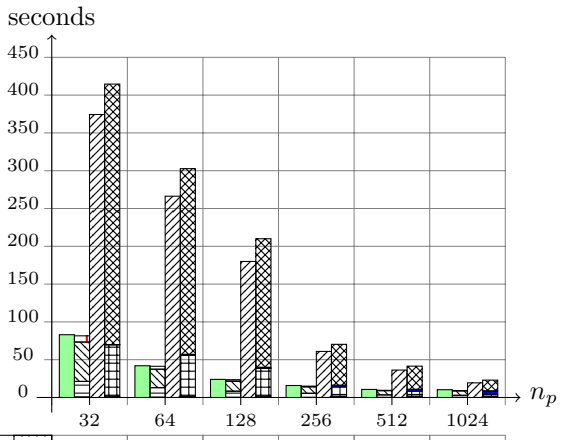


	32	64	128	256	512	1024
Coarsening	0.594	0.299	0.167	0.107	0.092	0.135
Balancing	20.53	12.58	8.08	5.29	3.27	2.75
Meshing	53.35	24.66	13.27	10.23	5.62	5.87
R-setup	8.27	4.03	1.87	1.19	0.506	0.242
Total Setup	**84.21**	**42.24**	**24.12**	**17.78**	**10.64**	**10.24**
LU	1.68	1.67	1.67	1.65	1.66	1.65
R + P	106.92	82.76	63.45	21.91	8.45	3.09
Scatter	0.057	1.733	2.67	49.31	3.27	4.74
FE Matvecs	521.08	357.0	268.65	96.41	38.09	14.78
Total Solve	**569.4**	**390.97**	**291.66**	**106.63**	**45.46**	**21.62**
Meshes	9	10	11	12	12	14

FIG. 4.5. *Fixed-size scalability for solving the constant coefficient Poisson problem (Equation 4.1) on an octree with 31.9M elements generated from a gaussian distribution of points. On the finest multigrid level, the coarsest octant was at level 3 and the finest octant was at level 17. 8 Multigrid levels were used. 8 iterations were required to reduce the 2-norm of the residual by a factor of* 10^{-14}. *702 Matvecs, 108 of which are on the finest level, were required. The total number of meshes generated for each case is also reported. Note that due to the addition of auxillary meshes, the total number of meshes is greater than the number of multigrid levels.*

load imbalance. Although, the process of constructing coarser meshes from a fine mesh is harder than iterative global refinements of a coarse mesh to generate a sequence of fine meshes; this is more practical since the fine mesh can be defined naturally depending on modeling restrictions, and/or physics of the problem as opposed to the coarse mesh, which is purely an artifact of the numerical method. It is also natural and more desirable to be able to control the fine mesh in an adaptive algorithm rather than controlling the coarse mesh.

- We have demonstrated good scalability of our implementation and can solve problems with billions of elements on thousands of processors in less than 10

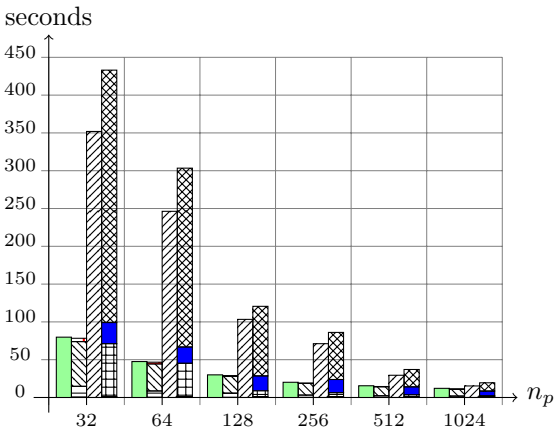


		32	64	128	256	512	1024
Coarsening		0.59	0.293	0.163	0.108	0.091	0.133
Balancing		20.69	12.69	8.174	5.468	3.34	2.76
Meshing		52.1	24.43	13.06	8.504	5.597	5.927
R-setup		8.224	3.87	1.82	0.95	0.487	0.243
Total Setup		**83.08**	**42.02**	**23.99**	**15.87**	**10.68**	**10.31**
LU		1.67	1.66	1.66	1.65	1.65	1.64
R + P		68.2	54.35	36.85	12.21	6.57	2.83
Scatter		0.055	1.04	1.39	2.29	2.52	4.17
FE Matvecs		344.91	245.85	170.22	54.22	30.76	14.34
Total Solve		**374.57**	**266.44**	**180.09**	**60.98**	**36.29**	**19.36**

FIG. 4.6. *Fixed-size scalability for solving the variable-coefficient Poisson problem (Equation 4.2) on the meshes used in Figure 4.5. 8 Multigrid levels were used. 5 iterations were required to reduce the 2-norm of the residual by a factor of* 10^{-10}. *468 Matvecs, 72 of which are on the finest level, were required.*

minutes. However, load balancing remains an open problem and this begins to affect our iso-granular scalability beyond a thousand processors. This is a difficult problem to tackle because there are many competing factors: Restriction, prologation, scatters and MatVecs.

- We have demonstrated that our implementation works well even on problems with variable coefficients.
- We have compared our geometric multigrid implementation with the algebraic multigrid scheme (BoomerAMG) implemented in a standard off-the-shelf package (HYPRE) and show that the proposed algorithm is quite competent. The setup cost for the matrix-free geometric multigrid algorithm is much lower than its algebraic multigrid counterpart. This makes it better suited for problems in which the linear system of equations is constructed and solved numerous times. Examples of such problems include time-dependent problems, non-linear problems and applications with adaptive mesh refinement proceedures.
- Our MPI-based implementation, DENDRO, is built on top of PETSc [6, 7]. Dendro is an open source code that can be downloaded from [46].



	32	64	128	256	512	1024
Coarsening	0.41	0.21	0.12	0.083	0.084	0.123
Balancing	14.37	8.44	5.63	3.11	2.19	1.78
Meshing	59.19	35.81	22.06	15.36	11.93	9.04
R-setup	4.4	2.26	1.14	0.58	0.33	0.17
Total Setup	**79.85**	**47.55**	**29.96**	**20.14**	**15.53**	**12.11**
LU	1.1	1.11	1.1	1.1	1.1	1.1
R + P	70.16	44.38	7.86	5.29	2.56	1.33
Scatter	27.95	21.38	19.63	17.43	10.42	6.14
FE Matvecs	334.1	236.7	92.08	62.33	22.99	10.89
Total Solve	**352.14**	**246.43**	**103.4**	**71.22**	**29.42**	**15.27**

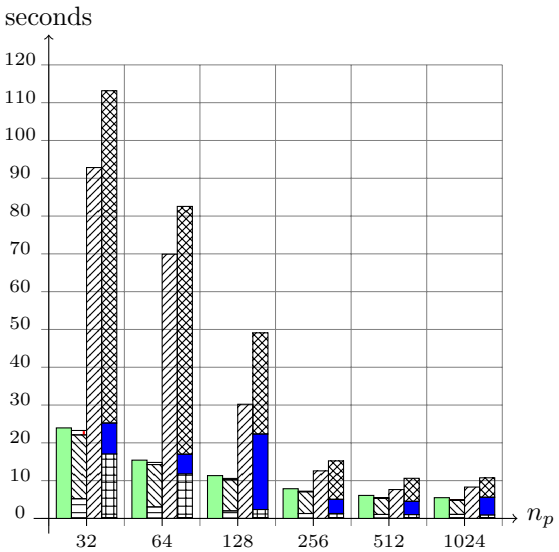
FIG. 4.7. *Fixed-size scalability for solving the variable-coefficient Poisson problem (Equation 4.2) on an octree with 22.4M elements generated using a log-normal distribution of points located on two diagonally opposite corners of the unit cube. 8 Multigrid levels and 15 meshes were used. 5 iterations were required to reduce the 2-norm of the residual by a factor of* 10^{-10}.

There are two important extensions for the present work: higher-order discretizations and integration with domain-decomposition methods such as the Heirarchical Hybrid Grids (HHG) scheme described in [10]. The former will result in improved accuracy with fewer elements and the latter will help solve problems involving complicated geometries with fewer elements. The last point stems from the fact that using a single octree to mesh a domain is more restrictive than allowing the use of multiple octrees, each of which is only responsible for a part of the entire domain.

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, and DMS-0612578. Computing resources on the TeraGrid's NCSA Intel 64 Linux cluster Abe were provided under the grant ASC070050N. Computing resources on the Cray XT3 system Bigben at the Pittsburgh Supercomputing Center were provided under the MCA04T026 award.

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



	32	64	128	256	512	1024
Coarsening	0.11	0.057	0.043	0.042	0.064	0.13
Balancing	5.1	2.99	1.96	1.26	0.99	0.94
Meshing	16.89	11.21	8.24	5.75	4.3	3.75
R-setup	1.17	0.56	0.34	0.16	0.11	0.079
Total Setup	**23.95**	**15.42**	**11.33**	**7.86**	**6.09**	**5.49**
LU	0.18	0.18	0.18	0.18	0.18	0.18
R + P	16.89	11.62	2.18	1.13	0.84	0.75
Scatter	8.18	5.2	19.98	3.75	3.51	4.68
FE Matvecs	87.91	65.55	26.76	10.16	6.09	5.14
Total Solve	**92.83**	**69.92**	**30.21**	**12.58**	**7.64**	**8.3**

FIG. 4.8. *Fixed-size scalability for solving the variable-coefficient Poisson problem (Equation 4.2) on an octree with 5.64M elements generated using a log-normal distribution of points located on two diagonally opposite corners of the unit cube. 8 Multigrid levels and 15 meshes were used. 5 iterations were required to reduce the 2-norm of the residual by a factor of* 10^{-10}.

Proceedings of SC2004, The SCxy Conference series in high performance networking and computing, Pittsburgh, Pennsylvania, 2004. ACM/IEEE.

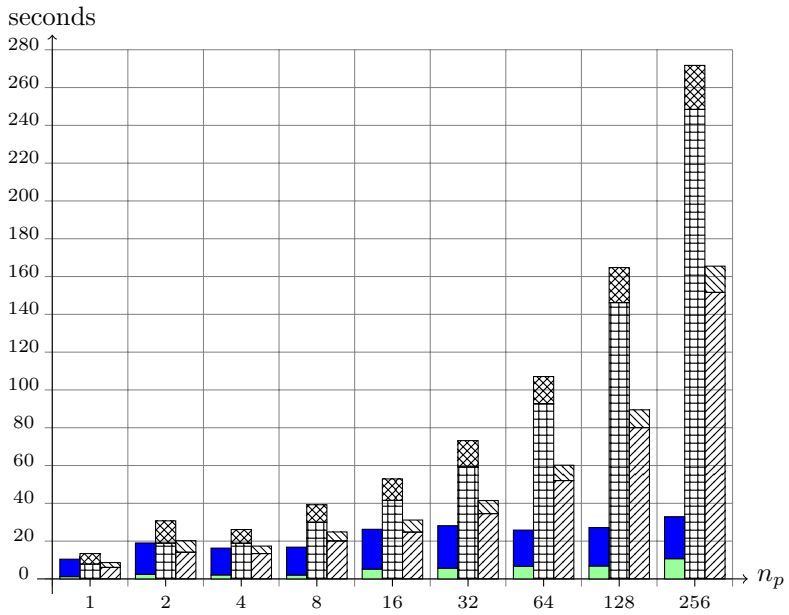
[2] Mark Adams and James W. Demmel. Parallel multigrid solver for 3d unstructured finite element problems. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 27, New York, NY, USA, 1999. ACM Press.

[3] 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.

[4] D. Ambrosi and F. Mollica. On the mechanics of a growing tumor. *International Journal of Engineering Science*, 40(12):1297–1316, 2002.

[5] 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*, pages 69–69, 1999.

[6] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc



	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

FIG. 4.9. *The variable-coefficient (contrast of 10^7) Poisson problem (Equation 4.3) was solved on meshes constructed on Gaussian distributions. A 7-level multiplicative-multigrid cycle was used as a preconditioner to CG for the GMG scheme. SuperLU_Dist was used to solve the coarsest grid problem in the GMG scheme. The coarsest multigrid level was distributed on a maximum of 32 processors in all experiments. For BoomerAMG, we experimented with two different coarsening schemes: Falgout coarsening and CLJP coarsening. The results from both experiments are reported. [33] reports that Falgout coarsening works best for structured grids and CLJP coarsening is better suited for unstructured grids. Since octree meshes lie in between both structured and generic unstructured grids, we compare our results using both the schemes. Both GMG and AMG schemes used 4 pre-smoothing steps and 4 post-smoothing steps per multigrid level with the damped Jacobi smoother. A relative tolerance of 10^{-10} in the 2-norm of the residual was used in all the experiments. The GMG scheme took about 12 CG iterations, the Falgout scheme took about 7 iterations and the CLJP scheme also took about 7 iterations. This experiment was performed on Abe. Each node of the cluster has 8 processors, which share an 8GB RAM. However, only 1 processor 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. 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 processors) compared to the time to setup the rest of the AMG scheme. The setup cost for the GMG scheme includes the time for constructing the mesh for all the multigrid levels (including the finest), constructing and balancing all the coarser multigrid levels, setting up the intergrid transfer operators by performing one restriction PreMatVec at each multigrid level and LU decomposition for the FE matrix at the coarsest grid. Note that the GMG scheme uses a matrix-free implementation and only the FE matrix for the coarsest grid is constructed explicitly. The time to construct and balance the finest octree is not included since it is a small overhead (≈ 2 seconds even on 256 processors) present in both the GMG and AMG schemes. The total time includes the setup time and the time required to solve the problem.*

users manual. Technical Report ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

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

[8] Randolph E. Bank and Todd Dupont. An optimal order process for solving finite element equations. *Mathematics of Computation*, 36(153):35–51, 1981.

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

[10] 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.

[11] 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.

[12] 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.

[13] D. Braess and W. Hackbusch. A new convergence proof for the multigrid method including the V-cycle. *SIAM Journal on Numerical Analysis*, 20(5):967–975, 1983.

[14] James H. Bramble, Joseph E. Pasciak, and Jinchao Xu. The analysis of multigrid algorithms for nonsymmetric and indefinite elliptic problems. *Mathematics of Computation*, 51(184):389–414, 1988.

[15] James H. Bramble, Joseph E. Pasciak, and Jinchao Xu. Parallel multilevel preconditioners. *Mathematics of Computation*, 55(191):1–22, 1990.

[16] Susanne C. Brenner and L. Ridgway Scott. *The mathematical theory of finite element methods*, volume 15 of *Texts in Applied Mathematics*. Springer-Verlag, New York, 1994.

[17] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A multigrid tutorial (2nd ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[18] 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.

[19] T. Corman, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

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

[21] J. E. Dendy. Black box multigrid. *Journal of Computational Physics*, 48:366–386, 1982.

[22] Robert Falgout, Andy Cleary, Jim Jones, Edmond Chow, Van Henson, Chuck Baldwin, Peter Brown, Panayot Vassilevski, and Ulrike Meier Yang. Hypre home page, 2001. http://acts.nersc.gov/hypre.

[23] Roland Glowinski, Tsorng-Whay Pan, Todd I. Hesla, Daniel D. Joseph, and Jacques Periaux. A fictitious domain method with distributed lagrange multipliers for the numerical simulation of particulate flow. *Contemporary Mathematics*, 218:121–137, 1998.

[24] Lena Gorelick, Meirav Galun, and Achi Brandt. Shape representation and classification using the poisson equation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(12):1991–2005, 2006. Member-Eitan Sharon and Member-Ronen Basri.

[25] 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.

[26] Michael Griebel and Gerhard Zumbusch. Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. *Parallel Comput.*, 25(7):827–843, 1999.

[27] David J. Griffiths. *Introduction to electrodynamics*. Prentice-Hall, New Jersey, 1999.

[28] David J. Griffiths. *Introduction to quantum mechanics*. Prentice-Hall, New Jersey, 2004.

[29] 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.

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

[31] Eldad Haber and Stefan Heldmann. An octree multigrid method for quasi-static Maxwell's equations with highly discontinuous coefficients. *J. Comput. Phys.*, 223(2):783–796, 2007.

[32] Wolfgang Hackbusch. *Multigrid methods and applications*, volume 4 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 1985.

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

[34] Brian Von Herzen and Alan H. Barr. Accurate triangulations of deformed, intersecting surfaces. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 103–110, New York, NY, USA, 1987. ACM Press.

[35] 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.

[36] M. Jung and U. rude. Implicit extrapolation methods for variable coefficient problems. *SIAM Journal on Scientific Computing*, 19(4):1109–1124, 1998.

[37] E. Kim, J. Bielak, O. Ghattas, and J. Wang. Octree-based finite element method for large-scale earthquake ground motion modeling in heterogeneous basins. *AGU Fall Meeting Abstracts*, pages B1221+, December 2002.

[38] Erwin Kreyszig. *Introductory functional analysis with applications*. John Wiley and Sons, Inc., New York, 1989.

[39] 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.

[40] 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.

[41] Jan Modersitzki. *Numerical Methods for Image Registration*. Numerical Mathematics and Scientific Computation. Oxford Univ Press, 2004.

[42] Maria Teresa Neves-Petersen and Steffen B. Petersen. Protein electrostatics: a review of the equations and methods used to model electrostatic equations in biomolecules–applications in biotechnology. *Biotechnology Annual Review*, 9:315–395, 2003.

[43] 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.

[44] John B. Pormann, Craig S. Henriquez, Jr. John A. Board, Donald J. Rose, David M. Harrild, and Alexandra P. Henriquez. Computer simulations of cardiac electrophysiology. In *Supercomputing '00: Proceedings of the IEEE/ACM SC2000 conference*. IEEE computer society, 2000.

[45] Isabelle Ramière, Philippe Angot, and Michel Belliard. A general fictitious domain method with immersed jumps and multilevel nested structured meshes. *Journal of Computational Physics*, 225(2):1347–1387, 2007.

[46] Rahul Sampath, Hari Sundar, Santi S. Adavani, Ilya Lashuk, and George Biros. `Dendro` home page, 2008. http://www.seas.upenn.edu/csela/dendro.

[47] Rahul Sampath, Hari Sundar, Santi S. Adavani, Ilya Lashuk, and George Biros. `Dendro` users manual. Technical report, University of Pennsylvania, 2008.

[48] Yair Shapira. Multigrid for locally refined meshes. *SIAM Journal on Scientific Computing*, 21(3):1168–1190, 1999.

[49] S. P. Spekreijse. Elliptic grid generation based on laplace equations and algebraic transformations. *Journal of Computational Physics*, 118(1):38–61, 1995.

[50] Hari Sundar, Rahul S. 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*, New York, NY, USA, 2007. ACM Press.

[51] 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*, (to appear), 2008.

[52] Osamu Tatebe and Yoshio Oyanagi. Efficient implementation of the multigrid preconditioned conjugate gradient method on distributed memory machines. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 194–203, New York, NY, USA, 1994. ACM.

[53] H. Tropf and H. Herzog. Multidimensional range search in dynamically balanced trees. *Angewandte Informatik*, 2:71–77, 1981.

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

[55] 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.

[56] Tiankai Tu, Hongfeng Yu, Leonardo Ramirez-Guzman, Jacobo Bielak, Omar Ghattas, Kwan-Liu Ma, and David R. O'Hallaron. From mesh generation to scientific visualization: an end-to-end approach to parallel supercomputing. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 91, New York, NY, USA, 2006. ACM Press.

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

[58] 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.

[59] Harry Yserentant. On the convergence of multi-level methods for strongly nonuniform families of grids and any number of smoothing steps per level. *Computing*, 30:305–313, 1983.

[60] Harry Yserentant. The convergence of multilevel methods for solving finite-element equations in the presence of singularities. *Mathematics of Computation*, 47(176):399–409, 1986.

[61] Shangyou Zhang. Optimal-order nonnested multigrid methods for solving finite element equations. I. On quasi-uniform meshes. *Mathematics of Computation*, 55(191):23–36, 1990.

[62] Shangyou Zhang. Optimal-order nonnested multigrid methods for solving finite element equations. II. On nonquasiuniform meshes. *Mathematics of Computation*, 55(192):439–450, 1990.

Appendix A. Proof showing that A_k is a symmetric positive operator w.r.t. $(\cdot,\cdot)_k$. Since V_k is a finite-dimensional normed space, every linear operator on V_k is bounded, in particular A_k is bounded. Since V_k is a finite-dimensional space, it is complete with respect to any norm defined on that space and in particular with respect to the norm induced by the inner-product under consideration. Hence, the space V_k along with the respective inner-product $(\cdot,\cdot)_k$ forms a Hilbert space [38]. Hence, A_k has a unique Hilbert-adjoint operator; in fact, as Equation A.1 shows A_k is also self-adjoint.

Equation 2.3, the coercivity of $a(u,v)$ and the symmetricity of $a(u,v)$ and $(\cdot,\cdot)_k$ together lead to Equation A.1.

$$\begin{aligned}(A_k v, v)_k &= a(v,v) > 0 \quad \forall v \neq 0 \in V_k \\ (A_k w, v)_k &= a(v,w) \\ &= (A_k v, w)_k \\ &= (w, A_k v)_k \quad \forall v, w \in V_k \end{aligned} \tag{A.1}$$

Appendix B. The prolongation matrix. Since the coarse-grid vector space is a subspace of the fine-grid vector space, any coarse-grid vector, v, can be expanded

independently in terms of the fine and coarse-grid basis vectors.

$$\begin{aligned} v &= \sum_{n=1}^{dim(V_{k-1})} v_{n,k-1}\phi_n^{k-1} \\ &= \sum_{m=1}^{dim(V_k)} v_{m,k}\phi_m^k \end{aligned} \tag{B.1}$$

In Equation B.1, $v_{n,k}$ and $v_{n,k-1}$ are the coefficients in the basis expansions for v on the fine and coarse grids, respectively. If we choose the standard finite element shape functions, then for each ϕ_i^k there exists a unique $p_i \in \Omega$ such that

$$\phi_j^k(p_i) = \delta_{ij} \quad \forall i, j = 1, 2, \ldots, dim(V_k) \tag{B.2}$$

In Equation B.2, δ_{ij} is the Kronecker delta function and p_i is the fine-grid vertex associated with ϕ_i^k. Equations B.1 and B.2 lead to

$$v_{i,k} = \sum_{j=1}^{dim(V_{k-1})} v_{j,k-1}\phi_j^{k-1}(p_i) \tag{B.3}$$

We can view the prolongation operator as a MatVec with the input vector as the coarse-grid nodal values (co-efficients in the basis expansion using the finite element shape functions as the basis vectors) and the output vector as the fine-grid nodal values. The matrix entries are then just the coarse-grid shape functions evaluated at the fine-grid vertices (Equation B.4).

$$\boxed{\mathbf{P_1}(i,j) = \phi_j^{k-1}(p_i).} \tag{B.4}$$

An equivalent formulation is to satisfy Equation 2.8 in the variational sense by taking an inner-product with an arbitrary fine-grid test function. This formulation also produces the vector of fine-grid nodal values as a result of a MatVec with the vector of coarse-grid nodal values and the matrix is defined by Equation B.5.

$$\boxed{\mathbf{P_2} = (\mathbf{M_k^k})^{-1}\mathbf{M_{k-1}^k}} \tag{B.5}$$

where,

$$\mathbf{M_{k-1}^k}(i,j) = (\phi_i^k, \phi_j^{k-1})_k. \tag{B.6}$$

Since the two formulations are equivalent, we have

$$\mathbf{P_1} = \mathbf{P_2}. \tag{B.7}$$

Appendix C. Derivation of the Galerkin condition. Define the functional

$$F^k(v_k) = \frac{1}{2}(A_k v_k, v_k)_k - (f_k, v_k)_k \quad \forall v_k \in V_k \tag{C.1}$$

Since A_k is a symmetric positive operator w.r.t $(\cdot,\cdot)_k$, the solution u_k to the Equation 2.4 satisfies

$$u_k = \arg \min_{\forall v_k \in V_k} F^k(v_k) \tag{C.2}$$

This is simply the Ritz FEM formulation. In the multigrid scheme, we want to find

$$v_{k-1} = \arg \min_{w_{k-1} \in V_{k-1}} F^k(v_k + Pw_{k-1}) \tag{C.3}$$

Here, P is the prolongation operator defined in Section 2.2.

$$\begin{aligned} F^k(v_k + Pw_{k-1}) = & \\ & \frac{1}{2}((A_k v_k + A_k P w_{k-1}), (v_k + P w_{k-1}))_k - (f_k, v_k + P w_{k-1})_k \\ = & \quad \frac{1}{2}(A_k v_k, v_k)_k + \frac{1}{2}(A_k P w_{k-1}, v_k)_k + \frac{1}{2}(A_k v_k, P w_{k-1})_k \\ & \qquad + \frac{1}{2}(A_k P w_{k-1}, P w_{k-1})_k - (f_k, v_k)_k \\ & \qquad - \frac{1}{2}(f_k, P w_{k-1})_k - \frac{1}{2}(f_k, P w_{k-1})_k \\ = & \quad F^k(v_k) + \frac{1}{2}(A_k P w_{k-1}, v_k)_k + \frac{1}{2}((A_k v_k - f_k), P w_{k-1})_k \\ & \qquad - \frac{1}{2}(f_k, P w_{k-1})_k + \frac{1}{2}(P^* A_k P w_{k-1}, w_{k-1})_{k-1} \end{aligned} \tag{C.4}$$

Here, P^* is the Hilbert adjoint operator of P with respect to the inner-products considered. Since, A_k is symmetric with respect to $(\cdot,\cdot)_k$ and since the vector spaces are real we have,

$$\frac{1}{2}(A_k P w_{k-1}, v_k)_k = \frac{1}{2}(P w_{k-1}, A_k v_k)_k = \frac{1}{2}(A_k v_k, P w_{k-1})_k \tag{C.5}$$

Hence, we have

$$F^k(v_k + Pw_{k-1}) = F^k(v_k) + F_G^{k-1}(w_{k-1}) \tag{C.6}$$

with F_G^{k-1} defined by

$$F_G^{k-1}(v_{k-1}) = \frac{1}{2}(A_{k-1}^G v_{k-1}, v_{k-1})_{k-1} - (f_{k-1}^G, v_{k-1})_{k-1}. \tag{C.7}$$

A_{k-1}^G and f_{k-1}^G are defined by Equation 2.12 (The "*Galerkin*" condition). Equations C.3 and C.6 together lead to

$$v_{k-1} = \arg \min_{w_{k-1} \in V_{k-1}} F_G^{k-1}(w_{k-1}) \tag{C.8}$$

Equation C.9 shows that A_{k-1}^G is symmetric with respect to $(\cdot,\cdot)_{k-1}$ and Equation C.10 shows that it is also positive.

$$\begin{aligned} (A_{k-1}^G u, v)_{k-1} &= (A_k P u, P v)_k \\ &= (P u, A_k P v)_k \\ &= (u, A_{k-1}^G v)_{k-1} \quad \forall u, v \in V_{k-1} \end{aligned} \tag{C.9}$$

$$
\begin{aligned}
(A^G_{k-1}u, u)_{k-1} &= (A_k Pu, Pu)_k \quad \forall u \in V_{k-1} \\
\forall u \in V_{k-1}, \quad &\exists\ w_u \in V_k \ \mid \ Pu = w_u \\
\Rightarrow (A^G_{k-1}u, u)_{k-1} &= (A_k w_u, w_u)_k \geq 0 \quad \forall u \in V_{k-1}
\end{aligned} \tag{C.10}
$$

Hence, the solution v_{k-1} to Equation 2.11 satisfies Equation C.8.

Appendix D. Restriction Matrix. Any fine-grid vector, w, and coarse-grid vector, v can be expanded in terms of the fine and coarse grid basis vectors respectively

$$
\begin{aligned}
w &= \sum_{m=1}^{dim(V_k)} w_m \phi^k_m \\
v &= \sum_{n=1}^{dim(V_{k-1})} v_n \phi^{k-1}_n
\end{aligned} \tag{D.1}
$$

Now, let

$$
R\phi^k_m = \sum_{l=1}^{dim(V_{k-1})} \mathbf{R}(l,m)\phi^{k-1}_l \quad \forall m = 1, 2, \ldots, dim(V_k) \tag{D.2}
$$

Using the definition of the restriction operator (Equation 2.13), we have

$$
\begin{aligned}
(R\phi^k_m, \phi^{k-1}_n)_{k-1} &= \sum_{l=1}^{dim(V_{k-1})} \mathbf{R}(l,m)(\phi^{k-1}_l, \phi^{k-1}_n)_{k-1} \\
&= (\phi^k_m, \phi^{k-1}_n)_k, \\
&\quad \forall m = 1, 2, \ldots, dim(V_k) \\
&\quad \forall n = 1, 2, \ldots, dim(V_{k-1})
\end{aligned} \tag{D.3}
$$

Thus,

$$
\boxed{\mathbf{R} = (\mathbf{M^{k-1}_{k-1}})^{-1}\mathbf{M^{k-1}_k}} \tag{D.4}
$$

where,

$$
\mathbf{M^{k-1}_k}(i,j) = (\phi^{k-1}_i, \phi^k_j)_k = \mathbf{M^k_{k-1}}(j,i) \tag{D.5}
$$

Appendix E. An equivalent formulation for the multigrid scheme. The coarse-grid operator defined in Equation 2.12 is expensive to build. Here, we will show that this operator is equivalent to the coarse-grid version of the operator defined in Equation 2.3. This operator can be implemented efficiently using a matrix-free scheme.

Using Equations 2.6, B.5, 2.12, and 2.14 we have

$$
\begin{aligned}
\mathbf{A^G_{k-1}} &= (\mathbf{M^{k-1}_{k-1}})^{-1}\tilde{\mathbf{A}}^{\mathbf{G}}_{\mathbf{k-1}} \\
\tilde{\mathbf{A}}^{\mathbf{G}}_{\mathbf{k-1}} &= \mathbf{M^{k-1}_k}(\mathbf{M^k_k})^{-1}\tilde{\mathbf{A}}_{\mathbf{k}}(\mathbf{M^k_k})^{-1}\mathbf{M^k_{k-1}}
\end{aligned} \tag{E.1}
$$

Since $V_{k-1} \subset V_k$, we can expand the coarse-grid basis vectors in terms of the fine-grid basis vectors as follows:

$$\phi_j^{k-1} = \sum_{i=1}^{dim(V_k)} \mathbf{c}(i,j)\phi_i^k \quad \forall j = 1, 2, \ldots, dim(V_{k-1}) \tag{E.2}$$

By taking inner-products with arbitrary fine-grid test functions on either side of Equation E.2, we have

$$\begin{aligned}(\phi_l^k, \phi_j^{k-1})_k = \sum_{i=1}^{dim(V_k)} \mathbf{c}(i,j)(\phi_l^k, \phi_i^k)_k \\ \forall j = 1, 2, \ldots, dim(V_{k-1}) \\ \forall l = 1, 2, \ldots, dim(V_k)\end{aligned} \tag{E.3}$$

This leads to

$$\mathbf{c_k^{k-1}} = (\mathbf{M_k^k})^{-1}\mathbf{M_{k-1}^k} \tag{E.4}$$

Using Equations 2.15, E.1, E.2, and E.4 we can show that

$$\boxed{\mathbf{\tilde{A}_{k-1}} = \mathbf{\tilde{A}_{k-1}^G} \; ; \; \mathbf{A_{k-1}} = \mathbf{A_{k-1}^G}} \tag{E.5}$$

Note that the fine-grid problem defined in Equation 2.4, the corresponding coarse-grid problem (Equation 2.11) and the restriction operator (Equation 2.14) all require inverting a mass-matrix. This could be quite expensive. Instead, we solve the following problem on the fine-grid

$$\tilde{A}_k u_k = \tilde{f}_k \tag{E.6}$$

and solve the following corresponding coarse-grid problem

$$\mathbf{\tilde{A}_{k-1}e_{k-1}} = \mathbf{M_{k-1}^{k-1}f_{k-1}^G} = \mathbf{\tilde{R}r_k} = \mathbf{r_{k-1}} \tag{E.7}$$

for the coarse-grid representation of the error, e_{k-1}, using the fine-grid residual, r_k, after a few smoothing iterations. Here, $\tilde{R}$ is the modified restriction operator, which can be expressed as

$$\mathbf{\tilde{R}} = \mathbf{M_{k-1}^{k-1}R}(\mathbf{M_k^k})^{-1} \tag{E.8}$$

Note, that this operator is the matrix-transpose of the prolongation operator derived using the variational formulation.

$$\boxed{\mathbf{\tilde{R}} = \mathbf{P_2}^T} \tag{E.9}$$

Since, $\mathbf{P_1} = \mathbf{P_2}$, we can use $\mathbf{P_1}^T$ instead of $\mathbf{\tilde{R}}$.

Appendix F. Minimum grain size required for good scalability. For good scalability of our algorithms, the number of elements in the interior of the processor

domains must be significantly greater than the number of elements on the inter-processor boundaries. This is because our communication costs are proportional to the number of elements on the inter-processor boundaries and by keeping the number of such elements small we can keep our communication costs low. Here we attempt to estimate the minimum grain size necessary to ensure that the number of elements in the interior of a processor is greater than those on its surface. In order to do this, we assume the octree to be a regular grid. Consider a cube which is divided into N^3 equal parts. There are $(N-2)^3$ small cubes in the interior of the large cube and $N^3-(N-2)^3$ small cubes touching the internal surface of the large cube. In order for the number of cubes in the interior to be more than the number of cubes on the surface, N must be $>= 10$. Hence, the minimum grain size per processor is estimated to be 1000 elements.