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 variable-coefficient elliptic partial differential equations on the unit box (with Dirichlet or Neumann boundary conditions) using highly nonuniform, octree-based, conforming finite element discretizations. Our octrees are 2:1 balanced, that is, we allow no more than one octree-level difference between octants that share a face, edge, or vertex. We describe a parallel algorithm whose input is an arbitrary 2:1 balanced fine-grid octree and whose output is a set of coarser 2:1 balanced octrees that are used in the multigrid scheme. Also, we derive matrix-free schemes for the discretized finite element operators and the intergrid transfer operations. The overall scheme is second-order accurate for sufficiently smooth right-hand sides and material properties; its complexity for nearly uniform trees is $\mathcal{O}(\frac{N}{n_p} \log \frac{N}{n_p}) + \mathcal{O}(n_p \log n_p)$, where N is the number of octree nodes and n_p is the number of processors. Our implementation uses the Message Passing Interface standard. We present numerical experiments for the Laplace and Navier (linear elasticity) operators that demonstrate the scalability of our method. Our largest run was a highly nonuniform, 8-billion-unknown, elasticity calculation using 32,000 processors on the Teragrid system, "Ranger," at the Texas Advanced Computing Center. Our implementation is publically available in the `Dendro` library, which is built on top of the `PETSc` library from Argonne National Laboratory.

Key words. geometric multigrid, meshing, 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

DOI. 10.1137/090747774

1. Introduction. Various physical and biological processes are modeled using elliptic operators such as the Laplacian operator. They are encountered in diffusive transportation of energy, mass and momentum [21], electromagnetism [29], quantum mechanics [30], models for tumor growth [4], protein folding and binding [43], and cardiac electrophysiology [45]. They are also used in nonphysical applications such as mesh generation [51], image segmentation [27], and image registration [42].

The finite element method is a popular technique for solving elliptic partial differential equations (PDEs) numerically. Finite element methods require grid generation (or meshing) to generate function approximation spaces. Regular grids are easy to generate but can be quite expensive when the solution of the PDE problem is highly localized. Localized solutions can be captured more efficiently using nonuniform or unstructured grids. However, the flexibility of unstructured grids comes at a price: they are difficult to construct in parallel, they are difficult to precondition, they incur the overhead of explicitly constructing element-to-node connectivity information, and

---

[†]School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA 30332 (rahulss@cc.gatech.edu).

[‡]Department of Biomedical Engineering and School of Computational Science and Engineering Division, Georgia Institute of Technology, Atlanta, GA 30332 (gbiros@acm.org).

they are generally cache inefficient because of random queries into this data structure [5, 31, 60]. Octree meshes seem like a promising alternative, at least for some problems [3, 10, 44]: they are more flexible than regular 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 [52].

Besides grid generation, an optimal solver is also necessary for scalability. Multigrid algorithms are known to be efficient for solving elliptic PDEs. They have been researched extensively in the past [7, 14, 15, 22, 32, 33, 50, 61, 62, 63, 64] and remain an active research area [1, 2, 10, 11, 24, 28, 32, 34]. A distinguishing feature of multigrid algorithms is that their convergence rate does not deteriorate with increasing problem size [18, 33, 57]. Some multigrid implementations even obtain optimal complexity by combining this feature with an operation count linear in the number of unknowns.

Multigrid algorithms can be classified into two categories: (a) geometric and (b) algebraic. The primary difference is that algorithms of the former type use an underlying mesh for constructing coarser multigrid levels (*coarsening*), and algorithms of the latter type use the entries of the fine-grid matrix for coarsening in a black-box fashion. Algebraic multigrid methods are gaining prominence due to their generality and the ability to deal with unstructured meshes. Geometric multigrid methods are less general but have low overhead, are quite fast, and are easy to parallelize (at least for structured grids). For these reasons, geometric multigrid methods have been quite popular for solving smooth, coefficient nonoscillatory elliptic PDEs on structured grids. Examples of state-of-the-art scalable parallel algebraic multigrid packages include `Hypre` [23] and `Trilinos` [25].

Contributions. In this work,[1] we propose a parallel bottom-up geometric multigrid algorithm on top of the 2:1 balancing and meshing algorithms [52, 53, 47] 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 among successive multigrid levels in parallel is a challenging task because the coarse and fine grids may have been partitioned across processors in completely different ways. In the present work, we describe a scalable, matrix-free implementation of the intergrid transfer operators.

---

[1] A short version of the new methodology has appeared in [52] (meshing) and [47] (`Dendro` library) but without discussion on the restriction and prolongation operators, without full algorithmic details, and with limited discussion on scalability.

[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 Krylov solvers.

Table 1.1
As an example of the capabilities of the proposed methodology, we report isogranular scalability for solving a linear elastostatics problem on a set of octrees with a grain size (on the finest multigrid level) of $80K$ (approx) elements per processor generated using a Gaussian distribution of points. A relative tolerance of 10^{-10} in the 2-norm of the residual was used. Eleven iterations were required in each case to solve the problem to the specified tolerance. This experiment was performed on "Ranger." The size of the problem is indicated in the "Elements" row, the "Max/Min Elements" row gives the load imbalance across processors, the "MG Levels" row indicates the number of multigrid levels (it differs from the number of "Meshes" because our algorithm duplicates meshes to allow for incompatible partitioning), "$R + P$" indicates restriction and prolongation costs, and "LU" is the coarse-grid solve. In the "Theory Row," we report an estimate of the time required using the asymptotic analysis complexity using constants fitted by the runs on $12 – 12288$ processors. The fine-level input octrees are highly nonuniform. The largest octants are at tree-level three, and the smallest octants are at a tree-level reported in the "Finest Octant's Level" row.

CPUs	12	48	192	768	3072	12288	32000
Coarsening	0.85	1.27	2.18	3.09	4.17	6.28	4.01
Balancing	1.76	2.08	2.96	3.87	7.19	13.33	12.97
Meshing	2.41	3.21	6.53	7.4	21.06	33.41	34.03
R-setup	0.27	0.32	0.323	0.32	0.35	0.54	0.63
Total Setup	**4.64**	**5.95**	**11.04**	**12.85**	**31.89**	**54.69**	**70.14**
(Theory)	(10.24)	(10.36)	(10.69)	(12.34)	(20.29)	(57.18)	(144.99)
LU	0.025	0.02	0.517	0.048	4.92	0.019	0.041
R + P	6.22	7.89	12.87	13.8	14.87	56.2	30.16
Scatter	3.9	6.1	13.83	15.26	20.63	62.61	50.76
FE MatVecs	145.8	166.73	174.27	173.64	186.11	221.01	169.81
Total Solve	**152.36**	**175.2**	**187.27**	**188.38**	**212.04**	**242.37**	**208.06**
(Theory)	(152.35)	(169.13)	(185.13)	(201.06)	(217.24)	(232.69)	(244.25)
Meshes	13	16	22	25	27	29	30
MG Levels	10	11	12	14	15	16	16
Elements	986.97K	3.97M	15.87M	63.4M	253.8M	1.01B	2.64B
Max/Min Elements	1.66	1.9	2.44	2.43	2.73	2.88	2.89
MPI_Wait	53.95	80.23	97.86	88.1	97.83	173.82	118.1
Finest Octant's Level	14	15	17	18	19	20	20

- 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. An example of such scaling can be found in Table 1.1. `Dendro` is an open-source code that can be downloaded from [48]. `Dendro` is tightly integrated with `PETSc` [6].

Limitations. Some of the limitations 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 [52] to support higher-order discretizations or by using an extrapolation technique such as the one suggested in [35].
- Problems with complex geometries are not directly supported in our implementation. In principle, `Dendro` can be combined with fictitious domain methods [26, 46] 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.
- Far-field and periodic boundary conditions are not supported in our implementation.

- The problem of load balancing across processors has not been fully addressed in this work.

Related work. We review only some of the recent work on multigrid methods for nonuniform meshes. In [13], a sequential geometric multigrid algorithm was used to solve two- and three-dimensional linear elastic problems using finite elements on nonnested unstructured triangular and tetrahedral meshes, respectively. The implementation of the intergrid transfer operations described in that work can be quite expensive for large problems and is nontrivial to parallelize. A sequential multigrid scheme for finite element simulations of nonlinear problems on quadtree meshes was described in [34]. In addition to the 2:1 balance 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 nonconforming discretizations. Projection schemes require two additional tree-traversals per MatVec, which we avoid in our approach. Multigrid algorithms for quadtree/octree meshes were also used in [10, 9, 44]. Reference [44] created the multigrid hierarchy using a simple coarsening strategy in which only the octants at the finest level were coarsened at each stage. While that coarsening strategy ensures that the 2:1 balance constraint is automatically preserved after each stage of coarsening, the decrease in the number of elements after coarsening might be small. An alternate coarsening strategy that tries to coarsen all octants was used in [10, 9] and in the present work. Reference [10] describes a sequential multigrid algorithm, and the corresponding parallel extension is described in [9]. In [9] a sequential graph-based scheme was used to partition the meshes on a dedicated master processor, and the resulting partitioned meshes were handed out to client processors, which performed the parallel multigrid solves. Hence, the scalability of their implementation was limited by the amount of memory available on the master processor. Moreover, the partitioning can be more expensive than the parallel computation of the solution.

Simpler scalable partitioning schemes based on space-filling curves have been used in [19, 28, 41] and in the present work. All these algorithms work on adaptive hierarchical Cartesian grids, which are constructed by the recursive refinement of grid cells into a fixed number of congruent subcells. In the approach used in [19, 41], each refinement produced three subcells in each coordinate direction. In the approach used in the present work and in [28], each refinement produces two subcells in each coordinate direction, and so the number of elements grows slower in this approach compared to the former approach. References [19, 28] used the additive version of multigrid, which is simpler to parallelize compared to the multiplicative version of multigrid used in the present work. However, the multiplicative version is considered to be more robust than the additive version as far as convergence rates are concerned [8]. A parallel multiplicative multigrid algorithm for nonuniform meshes was presented in [38]; that work reported good scalability results on up to 512 processors. In [38], the smoothing at each grid was performed only in the refined regions and in a small neighborhood around the refined regions. In contrast, we chose to cover the entire domain at each grid, and this allows us to use a simpler scheme to distribute the load across processors. A three-dimensional parallel algebraic multigrid method for unstructured finite element problems was presented in [2], which was based on 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 [11], a calculation with over 11 billion elements was reported. The authors proposed a scheme for conforming discretizations and geo-

metric multigrid solvers on semistructured meshes. Their approach is highly scalable for nearly structured meshes, but it somewhat limits adaptivity because it is based on regular refinement of a coarse grid.

Additional examples of scalable approaches for nonuniform 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. The high costs related to partitioning, setup, and accessing generic unstructured grids has motivated the design of octree-based data structures. A characteristic feature of octree meshes is that they contain *hanging* vertices. In [52], we presented a strategy to tackle these hanging vertices and build conforming, trilinear finite element discretizations on these meshes.

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 [7, 15, 16, 17, 62, 63, 64]. 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. Instead of assembling the matrices, we implement a function that takes a vector as input and returns another vector that is the result of multiplying the matrix with the input vector; this function is referred to as a MatVec. We describe how we handle hanging vertices in our MatVec implementations. In section 4, we present the results from fixed-size and isogranular scalability experiments.

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

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

and the appropriate boundary conditions on the boundary of the domain $\partial\Omega$. This problem has a unique solution (see [17]).

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 (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. Then the discretized problem is to find an approximation of u, $u_k \in V_k$ such that

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

The discretized problem has a unique solution, and the sequence $\{u_k\}$ converges to u [17].

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

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

---

[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.

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

$$(2.4) \qquad\qquad\qquad A_k u_k = f_k,$$

where $f_k \in V_k$ is defined by

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

The operator A_k is a symmetric (self-adjoint) positive operator with respect to $(\cdot, \cdot)_k$. (In the following sections, we use italics to represent an operator (or vector) in the continuous form and use boldface to represent the matrix (or vector) corresponding to its co-ordinate basis representation.)

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

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

(2.6)

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

2.2. Prolongation. The prolongation operator is a linear operator

$$(2.7) \qquad\qquad\qquad P : V_{k-1} \rightarrow V_k$$

defined by

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

This is a standard prolongation operator and has been used before [17, 18]. The variational form of (2.8) is given by

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

In Appendix B, we show that

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

In (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

$$(2.11) \qquad\qquad A_{k-1}^G v_{k-1} = f_{k-1}^G,$$

where A_{k-1}^G and f_{k-1}^G are defined by the Galerkin condition (2.12) [18]

$$
\begin{aligned}
A_{k-1}^G &= P^* A_k P, \\
f_{k-1}^G &= P^*(A_k v_k - f_k) \\
&\qquad \forall v_{k-1} \in V_{k-1}, v_k \in V_k.
\end{aligned}
$$

(2.12)

ALGORITHM 1. TWO-GRID CORRECTION SCHEME

1. Relax ν_1 times on (E.6) with an initial guess, u_k^0.   (Presmoothing)

2. Compute the fine-grid residual using the solution vector, v_k, at the end of the presmoothing 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 (E.7).   (Coarse-grid correction)

5. Correct the fine-grid approximation: $\mathbf{v_k^{new}} = \mathbf{v_k} + \mathbf{P}\mathbf{e_{k-1}}$.   (Prolongation)

6. Relax ν_2 times on (E.6) with the initial guess, v_k^{new}.   (Postsmoothing)

Here P is the prolongation operator defined in section 2.2, and P^* is the Hilbert adjoint operator[4] of P with respect to the inner products $(\cdot, \cdot)_k$ and $(\cdot, \cdot)_{k-1}$.

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

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

In Appendix D, we show that

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

where

$$(2.15) \qquad \mathbf{M}_{\mathbf{k}}^{\mathbf{k-1}}(i,j) = (\phi_i^{k-1}, \phi_j^k)_k = \mathbf{M}_{\mathbf{k-1}}^{\mathbf{k}}(j,i).$$

2.5. A note on implementing the operators. The fine-grid operator A_k, the coarse-grid operator A_{k-1}^G, and the restriction operator R are expensive to implement using (2.6), (2.12), and (2.14), respectively. 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 $\mathbf{P}^T$ ((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 [17, 18].

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 [52] 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.

---

[4]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 [37].

3.1. Review of octrees. An octree[5] is a tree data structure that is used for spatial decomposition. Every node[6] 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*. The depth of an octant from the root is referred to as its *level*.[7] We use a *linear* octree representation (i.e., we exclude interior octants) using the *Morton encoding* scheme [12, 20, 52, 53, 56, 58]. Any octant in the domain can be uniquely identified by specifying one of its vertices, also known as its *anchor*, and its level in the tree. By convention, the anchor of an octant is its front lower left corner (a_0 in Figure 3.1). An octant's configuration with respect to its parent is specified by its *child number*: the octant's position relative to its siblings in a sorted list. The child number of an octant is a function of the coordinates of its anchor and its level in the tree. For convenience, we use the Morton ordering to number the vertices of an octant. Hence, an octant with a child number equal to k will share its kth vertex with its parent. This is a useful property that will be used frequently in the remaining sections. In order to perform finite element calculations of octrees, we impose a restriction on the relative sizes of adjacent octants. This is known as the 2:1 *balance constraint* (not to be confused with load balancing): No octant should be more than twice the size of any other octant that shares a corner, edge, or face with this octant. In this paper, we use the balancing algorithm described in [53]. *In this work, we deal with only sorted, complete, linear, 2:1 balanced octrees.*

3.2. Finite elements on octrees. In our previous work [53, 52], we developed low-cost algorithms and efficient data structures for constructing conforming finite element meshes using linear octrees. The key features of those algorithms are listed as follows:

- 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 the midpoints 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 finite element method solution.
- Since we handle hanging vertices in the meshing stage itself, we don't need to use projection schemes like those used in [3, 34, 36, 59] 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 precomputed 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

---

[5]Sometimes, we use quadtrees for illustration purposes. Quadtrees are two-dimensional analogues of octrees.

[6]The term *node* is usually used to refer to the vertices of elements in a finite element mesh; in the context of tree data structures, it refers to the octants themselves.

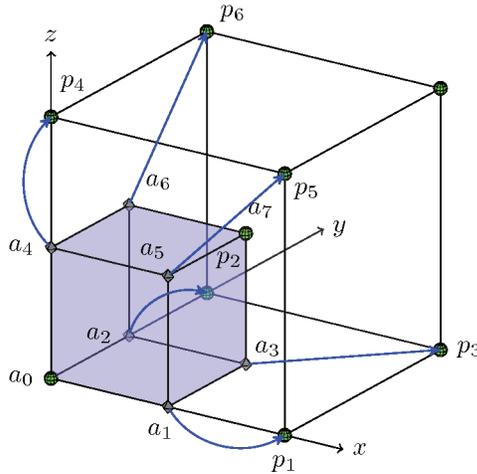[7]This is not to be confused with the term *multigrid level*.

FIG. 3.1. *Illustration of nodal connectivity required to perform conforming finite element method calculations using a single tree traversal. Every octant has at least two nonhanging vertices, one of which is shared with the parent and the other is shared amongst all the siblings. The shaded octant* (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* $a_3, a_5,$ *and* a_6 *are face-hanging vertices and point to* $p_3, p_5,$ *and* $p_6,$ *respectively. Similarly* $a_1, a_2,$ *and* a_4 *are edge-hanging vertices and point to* $p_1, p_2,$ *and* $p_4,$ *respectively. All the vertices in this illustration are labeled in the Morton ordering.*

the element-to-vertex mappings can be compressed at a modest expense of uncompressing 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:

- No shape functions are rooted at 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 nonhanging vertices in the octree.
- The support of a shape function can spread over more than eight elements.
- If a vertex of an element is hanging, then the shape functions rooted at the other nonhanging vertices in that element do not vanish on this hanging vertex. Instead, they will vanish at the nonhanging vertex to which this hanging vertex is mapped. If the ith vertex of an element/octant is hanging, then the index corresponding to this vertex will point to the ith vertex of the parent[8] of this element instead. For example, in Figure 3.1, 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 nonzero values within an octant if and only if it is rooted at some nonhanging vertex of this octant or if some vertex of the octant under consideration is hanging, say, the ith vertex, and the shape function in question is rooted at the ith nonhanging vertex of the parent of this octant. Hence, for any octant there are exactly eight shape functions

---

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

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 [36, 58, 59].

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 help us reduce the total number of permissible hanging configurations. Figure 3.1 illustrates these properties.

- Every octant has at least two nonhanging vertices:
  - 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,
  - the vertex that is common to both this octant and its parent.

Any element in the mesh belongs to one of eight child number-based configurations. After factoring in the above constraints, there are only 18 potential hanging-vertex configurations for each child number configuration.

3.2.1. Overlapping communication with computation. Every octant is owned by a single processor. However, the values of unknowns associated with octants on interprocessor 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. This is used to select the appropriate element stencil from a list of precomputed 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.[9] Thus, there is only one communication phase within each MatVec, which we can overlap with a computation phase:

1. Initiate nonblocking 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.
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 kth octree is either present in the $k+1$th octree or all of its eight children are present (Figure 3.2).

We construct the kth 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 seven successive elements following a "Child-0" element is either

---

[9]This is only possible because, our meshing scheme also builds the element-to-node connectivity mappings for the appropriate ghost elements. Although, this adds an additional layer of complexity to our meshing algorithm, it saves us one communication per MatVec.
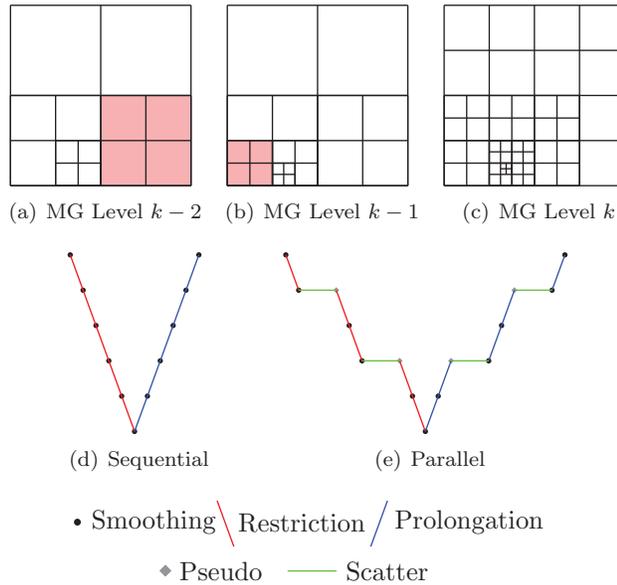
(a) MG Level $k-2$ (b) MG Level $k-1$ (c) MG Level k

(d) Sequential   (e) Parallel

• Smoothing \ Restriction / Prolongation

◆ Pseudo   —— Scatter

FIG. 3.2. (a)–(c) *Quadtree meshes for three successive multigrid levels. The shaded octants in (a) and (b) were not coarsened because doing so would have violated the 2:1 balance constraint. (d) A V-cycle where the meshes at all multigrid levels share the same partition. (e) A V-cycle where not all meshes share the same partition. Some meshes do share the same partition, and whenever the partition changes, a pseudomesh is added. The pseudomesh is used only to support intergrid transfer operations, and smoothing is not performed on this mesh.*

one of its siblings or a descendant 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 three cases: (a) $j < (i+8)$, (b) $j = (i+8)$, and (c) $j > (i+8)$. In the first case, the elements with indices in the range $[i, j)$ are not coarsened. In the second case, the elements with indices in the range $[i, j)$ are all leaves and 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.

One-level coarsening is an operation with $\mathcal{O}(N)$ work complexity, where N is the number of leaves in the $k+1$th octree. It is easy to parallelize and has an $\mathcal{O}(\frac{N}{n_p})$ parallel time complexity, where n_p is the number of processors.[10] The main parallel operations are two circular shifts: one clockwise and another counter-clockwise. The message in each case is just one 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 between the first and last Child-0 elements on each processor.

However, the operation described above may produce 4:1 balanced octrees[11] instead of 2:1 balanced octrees. Hence, we balance the result using the algorithm

---

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

[11]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.

described in [53]. 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.

The sequence of octrees constructed as described above has the property that nonhanging vertices in any octree remain nonhanging in all the finer octrees as well. Hanging vertices on any octree could either become nonhanging on a finer octree or remain hanging on the finer octrees, too. In addition, an octree can have new hanging as well as nonhanging 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 nonhanging fine-grid vertices that lie within the support of each coarse-grid shape function. This is trivial on regular grids, but for nonuniform grids, it 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 explicitly; 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 nonhanging fine-grid vertices that lie within this coarse-grid octant. These form the entries of the restriction and prolongation matrices (see (2.10)).

3.4.1. Alignment and partitioning of grids. To parallelize the intergrid transfer operations, 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 in both the coarse and fine grids, then the same processor must *own* this octant on both 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 eight 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, one problem with maintaining a single partition across all multigrid levels is that the coarser multigrid levels might be too sparse to be distributed across all the processors. As explained in section 3.6, we enforce a minimum grain size (elements per processor) for all grids, and this limits the number of processors used for each grid. Another reason to use different partitions for the different grids is to get better load distribution across the processors for the smoothing operation. Hence, we allow certain multigrid levels to be partitioned differently than others.[12] When a transition in the partitions is required, we duplicate the octree in question, and we let one of the duplicates share the same partition as that of its immediate finer multigrid level and let the other one

---

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

share the same partition as that of its immediate coarser multigrid level. We refer to one of these duplicates as the *pseudo mesh* (Figure 3.2). The pseudomesh is used only 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 redistributing the values from one partition to another. We also want to reduce the number of pseudomeshes in order to lower setup costs and to lower communication costs by avoiding scatter operations. Hence, we do the following checks to avoid pseudomeshes if possible:

- If a fine grid can use more processors than its immediate coarse grid, we first check the increase in average grain size if the fine grid used the same number of processors as the coarse grid. If this increase is small (we allow a factor of 1.5 in our implementation), we restrict the number of processors on the fine grid to be the same as that for the coarse grid.
- We check the load imbalance on each fine grid if it were to use the same partition as its immediate coarse grid. If this imbalance is below a user-specified threshold, we use the same partition for the fine grid and its immediate coarse grid.

3.4.2. MatVecs for restriction and prolongation. 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 make use of the following observations:

- Every nonhanging 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 nonhanging 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 nonhanging vertex. So the correct index is always stored immaterial of the hanging state of the vertex.

We 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 includes (a) the cost of a few bitwise operations for each fine-grid octant and (b) the memory bandwidth required for reading the eight-byte mask. The latter cost is comparable to the cost required for reading a material property array within the finite element MatVec (for a variable coefficient operator). The restriction and prolongation MatVecs are operations with $\mathcal{O}(N)$ work complexity and have an $\mathcal{O}(\frac{N}{n_p})$ parallel time complexity. The following section describes how we compute these masks for any given pair of coarse and fine octrees.

3.4.3. Computing the "masks" for restriction and prolongation. Each nonhanging fine-grid vertex has a maximum[13] of 1758 unique locations at which a coarse-grid shape function that contributes to this fine vertex could be rooted. 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 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 eight 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 nonhanging 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. Listing all possible constraints is unnecessary for our purposes; 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 nonhanging coarse-grid locations around any fine-grid vertex.

We cannot 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 performed only 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 possibilities 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.

---

[13]This is a weak upper bound.

3.4.4. 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 cannot 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 nonblocking 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. Receive 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 nonblocking 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 that in step 2, we iterated over only some of these elements. In this step, we iterate over the remaining elements.
7. Receive the ghost values sent from other processors in step 5.
8. Add the values received 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 [18, 22]. 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 multigrid 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. This is one reason why the convergence of the stand-alone multigrid solver deteriorates with increasing contrast in material properties. Coarsening across discontinuities also affects the coarse grid correction, even when the Galerkin condition is satisfied. Large contrasts in material properties also affect simple smoothers like the Jacobi smoother. The standard solution is to use multigrid as a preconditioner to the conjugate gradient (CG) method. 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. 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 interprocessor boundaries. This is because communication costs are proportional to the number of elements on the interprocessor boundaries, and, by keeping the number of such elements small, we can keep our communication costs low. We use a heuristic 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 that 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 greater than or equal to 10. Hence, the minimum grain size per processor is estimated to be 1000 elements.

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

1. A *sufficiently* fine[14] 2:1 balanced complete linear octree is constructed using the algorithms described in [53].
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 (section 3.6) is computed.
4. Starting with the coarsest octree, the octree at each multigrid level is meshed using the algorithm described in [52]. As long as the load imbalance across processors is acceptable and 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 grid without violating the minimum grain size criteria, the partition of the coarser grid is imposed onto the finer grid during meshing. If either of the above two conditions is violated, then the octree for the finer grid is duplicated. One of them is meshed using the partition of the coarser grid, and the other is meshed using a fresh partition. 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.

The discrete system of equations is solved using the CG algorithm preconditioned with the multigrid scheme.

4. Numerical experiments. In this section, we consider solving for $\mathbf{u}$ in (4.1) and u in (4.2), (4.3), (4.4), and (4.5). Equation (4.1) represents a three-dimensional, linear elastostatics (vector) problem with isotropic and homogeneous Lamé moduli (μ and λ) and homogeneous Dirichlet boundary conditions. Equations (4.2) through (4.5) represent three-dimensional, linear Poisson (scalar) problems with inhomogeneous material properties and homogeneous Neumann boundary conditions.

$$\mu\Delta\mathbf{u} + (\lambda+\mu)\nabla\,\mathrm{Div}\,\mathbf{u} = \mathbf{f}\ \text{in}\ \Omega,$$
$$\mathbf{u} = \mathbf{0}\ \text{in}\ \partial\Omega,$$
(4.1) $\qquad \mu = 1;\ \lambda = 4;\ \Omega = [0,1]^3,$
$$-\nabla\cdot(\epsilon\nabla u) + u = f\ \text{in}\ \Omega,$$
$$\hat{n}\cdot\nabla u = 0\ \text{in}\ \partial\Omega,$$
(4.2) $\qquad \epsilon(x,y,z) = \left(1 + 10^6\left(\cos^2(2\pi x) + \cos^2(2\pi y) + \cos^2(2\pi z)\right)\right),$
(4.3) $\qquad \epsilon(x,y,z) = \begin{cases} 10^7 & \text{if } 0.3 \leq x,y,z \leq 0.6, \\ 1.0 & \text{otherwise,} \end{cases}$

---

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

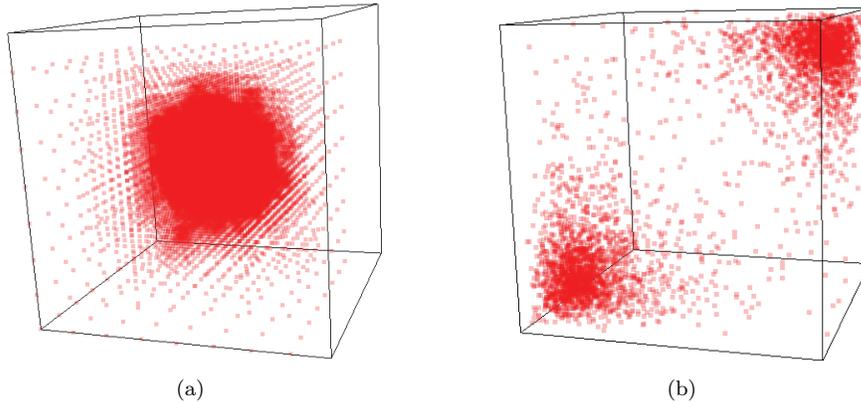(a)   (b)

FIG. 4.1. *Samples of the point distributions used for the numerical experiments:* (a) *a Gaussian 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 its mirror image about the main diagonal.*

$$(4.4) \qquad \epsilon(x,y,z) = \begin{cases} 10^7 & \text{if the index of the octant} \\ & \text{containing } (x,y,z) \text{ is divisible by} \\ & \text{some given integer } K, \\ 1.0 & \text{otherwise,} \end{cases}$$

$$(4.5) \qquad \epsilon(x,y,z) = \begin{cases} 10^7 & \text{if } (x,y,z) \in [0,0.5] \times [0,0.5] \times [0,0.5] \\ & \cup [0.5,1.0] \times [0.5,1.0] \times [0,0.5] \\ & \cup [0,0.5] \times [0.5,1.0] \times [0.5,1.0] \\ & \cup [0.5,1.0] \times [0,0.5] \times [0.5,1.0], \\ 1.0 & \text{otherwise.} \end{cases}$$

We discretized these problems on various octree meshes generated using Gaussian and log-normal distributions.[15] 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 ranges from about 25,000 to over 1 billion and were solved on up to 32000 processors on the Teragrid system "Ranger" (63,000 Barcelona cores with Infiniband). Details for this system can be found in [54]. Our C++ implementation uses MPI, PETSc [6], and SuperLU_Dist [39]. The runs were profiled using PETSc.

In this section, we present the results from four sets of experiments: (A) a convergence test, (B) a robustness test, (C) isogranular scalability, and (D) fixed size scalability. The parameters used in the experiments are listed below:
- 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) through (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 CG method in all experiments. This is known to be more robust than the stand-alone multigrid algorithm for variable-coefficient problems [55].

---

[15]In the following experiments, the octrees were not generated based on the underlying material properties. In [49], we give some examples for constructing octrees based on user-supplied data such as material properties and source terms.

TABLE 4.1

L^2 norm of the error between the true solution and its finite element approximation for the variable-coefficient problem (4.2). The sequence of meshes used in this experiment was 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.

Max. Element Size ($h_{\max}$)	1/16	1/32	1/64	1/128	1/256
L^2 Norm of the Error	3.98×10^{-3}	9.62×10^{-4}	2.46×10^{-4}	6.18×10^{-5}	1.56×10^{-5}

TABLE 4.2

The number of iterations required to reduce the 2-norm of the residual in (4.4) by a factor of 10^{-8} for different values of K, a parameter that controls the frequency of jumps. A regular grid with 128 elements in each dimension was used for this experiment.

$\log_2 K$	1	4	7	10	13	16	19
$Its.$	119	18	25	55	66	43	17

- The damped Jacobi method was used as the smoother at each multigrid level.
- SuperLU_Dist [39] was used to solve the coarsest-grid problem in all cases.
- In order to minimize communication costs, the coarsest grid used fewer processors than the finer grids. 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 (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[16] in the 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. Robustness test. In the second experiment, we tested the robustness of the multigrid solver in the presence of strong jumps in the material properties. We discretized (4.4) and (4.5) on a uniform octree with about 2M elements and measured the convergence rate for different values of K. Six multigrid levels were used for these problems. In Table 4.2, we report the number of iterations that were required to reduce the 2-norm of the residual in (4.4) by a factor of 10^{-8} for different values of K; the number of jumps decreases as K increases. It is apparent that the solver is quite sensitive to the number of jumps. However, there are other factors that determine the overall performance of the solver. For example, it only takes seven iterations to solve (4.5) to the same tolerance, although there are more jumps in (4.5) than in (4.4) for $\log_2 K = 10, 13, 16$, or 19. While the fine-grid material properties in (4.5) are represented exactly on all coarser grids, the fine-grid material properties in (4.4) are not represented accurately on any of the coarse grids. This would explain why coarse-grid correction works better for (4.5) than for (4.4). The results of this experiment show that the current scheme is not robust in the presence of discontinuous coefficients.

4.3. Parallel scalability results. We tested the scalability of our implementation on the TeraGrid system Ranger. In all the fixed-size (strong) and isogranular (weak) scalability results, the reported times for each component are the maximum values for that component across all the processors. Hence, in some cases the total

[16]Any element whose length is greater than $h_{\max}$.

time[17] is lower than the sum of the individual components. We also report the theoretical predictions[18] for the total setup and solve times. This was computed using the asymptotic complexity estimates for the setup $(\mathcal{O}(\frac{N}{n_p} \log \frac{N}{n_p}) + \mathcal{O}(n_p \log n_p))$ and solve $(\mathcal{O}(\frac{N}{n_p}) + \mathcal{O}(\log n_p))$ times. The coefficients in the expressions for the complexity were computed so that the sum of squares of the deviation between the theoretical estimates and the actual data is minimized. While determining these coefficients, we skipped the last data point (corresponding to the greatest number of processors) in each experiment. This was done so that we could use our model to predict the value for the last data point and compare our predictions with the observed results. The number of multigrid levels and the total number of meshes generated for each case are also reported. Note that due to the addition of auxiliary meshes, the total number of meshes is greater than the number of multigrid levels. The setup cost includes the time for constructing the mesh for all the multigrid levels (including the finest), constructing and balancing all the coarser multigrid levels, and setting up the intergrid transfer operators by performing one PreMatVec at each multigrid level. The time to create the work vectors for the multigrid scheme and the time to build the coarsest-grid matrix are also included in the total setup time but are not reported individually since they are insignificant. "Scatter" refers to the process of transferring the vectors between two different partitions of the same multigrid level during the intergrid transfer operations, required whenever the coarse and fine grids do not share the same partition. The time spent in applying the Jacobi preconditioner, computing the inner products within CG, and solving the coarsest grid problems using `SuperLU_Dist` are all accounted for in the total solve time but are not reported individually since they are insignificant. When we report `MPI_Wait()` times, we refer to synchronization for nonblocking operations during the restriction, prolongation, and finite element MatVecs.

4.3.1. 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 Tables 4.3, 1.1, and 4.4. Tables 4.3 and 1.1 report the results for the constant-coefficient elasticity problem (4.1) for two different grain sizes, and Table 4.4 reports the results for the variable-coefficient Poisson problem (4.2). The results from an isogranular scalability experiment for solving the variable-coefficient Poisson problem (4.2) on octrees generated from log-normal point distributions are reported in Table 4.5. There is little variation between the Gaussian distribution case and the log-normal distribution case. For the Gaussian distribution cases, the coarsest octant at the finest multigrid level was at level three; the level of the finest octant at the finest multigrid level for each case is reported in the tables. The octrees considered here are extremely nonuniform—roughly five orders of magnitude variation in the leaf size. It is also 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. This is evident from (a) the imbalance in the number of elements per processor and (b) the time spent in calls to `MPI_Wait()`. These numbers are reported in Tables 4.3 and 1.1

---

[17]This is reported in bold face.
[18]This is reported within parenthesis just below the total setup and solve times.

TABLE 4.3
Isogranular scalability for solving the constant coefficient linear elastostatics problem on a set of octrees with a grain size (on the finest multigrid level) of 30K (approximate) elements per CPU (n_p) generated using a Gaussian distribution of points. A relative tolerance of 10^{-10} in the 2-norm of the residual was used. Eleven iterations were required in each case to solve the problem to the specified tolerance.

CPUs	12	48	192	768	3072	12288	32000
Coarsening	0.33	0.56	0.95	1.66	2.18	4.39	1.90
Balancing	0.77	0.99	1.23	1.84	4.48	12.66	9.67
Meshing	0.973	1.44	1.82	3.32	15.09	32.89	21.67
R-setup	0.092	0.125	0.122	0.14	0.172	0.173	0.355
Total Setup	**2.41**	**3.22**	**3.87**	**6.51**	**23.4**	**53.21**	**47.14**
(Theory)	(4.94)	(4.97)	(5.25)	(7.02)	(15.48)	(54.86)	(148.39)
LU	0.189	0.4	0.017	0.171	0.543	0.015	0.0025
R + P	2.53	3.62	4.23	5.36	8.55	8.96	11.97
Scatter	3.11	6.49	8.59	13.13	16.98	21.15	27.88
FE MatVecs	51.63	57.73	60.20	64.58	68.78	69.87	66.91
Total Solve	**54.82**	**63.74**	**66.14**	**73.5**	**80.96**	**85.28**	**89.77**
(Theory)	(55.87)	(61.73)	(67.39)	(73.60)	(79.79)	(86.06)	(90.33)
Meshes	11	16	19	24	25	28	29
MG Levels	8	11	12	14	14	15	16
Elements	337.8K	1.34M	5.29M	21.15M	84.5M	338.3M	880.3M
Max/Min Elements	1.89	1.79	2.04	2.82	3.9	3.08	3.12
MPI_Wait	21.32	24.32	29.58	32.65	39.8	39.19	95.75
Finest Octant's Level	12	15	16	18	18	19	19

(in the Introduction).[19] 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 grids, especially for nonuniform distributions.
- Partitioning each multigrid level independently to get good load balance for the smoothing operations would require the creation of an auxiliary 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.

4.3.2. 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 solving the variable-coefficient problem (4.2) on an octree with 32M (approximate) elements generated from Gaussian point distribution are reported in Table 4.6. This experiment was repeated on octrees with 22M and 6M (approximate) elements generated using log-normal point distributions, and the corresponding results are reported in Tables 4.7 and 4.8, respectively. The results for the Gaussian and log-normal distributions are similar. We observe good speed ups for the setup phase up to 256 processors, and the

---

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

TABLE 4.4

Isogranular scalability for solving the variable-coefficient Poisson problem (4.2) *on the set of octrees with a grain size (on the finest multigrid level) of* $0.25M$ *elements (approximate) per processor* (n_p) *generated using a Gaussian distribution of points. The iterations were terminated when the* 2*-norm of the residual was reduced by a factor of* 10^{-10}. *Five iterations were required in each case. All timings are reported in seconds.*

CPUs	1	4	16	64	256	1024	4096
Coarsening	0.02	0.09	2.79	3.41	4.48	5.44	6.75
Balancing	0.34	2.32	4.66	5.23	6.18	7.96	8.92
Meshing	2.66	7.44	7.12	8.12	20.43	19.54	29.14
R-setup	0.48	1.04	0.85	0.88	1.01	0.99	1.04
Total Setup	**3.59**	**11.11**	**13.07**	**15.32**	**30.29**	**30.58**	**44.95**
LU	1.16	0.27	1.11	2.69	5.4	11.69	10.99
R + P	2.07	5.61	5.26	6.75	6.99	7.26	10.22
Scatter	0	0	0.11	0.32	2.55	3.65	6.79
FE MatVecs	20.37	43.92	37.46	39.97	40.64	40.96	52.53
Total Solve	**24.19**	**49.98**	**43.53**	**48.19**	**53.06**	**60.37**	**73.46**
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
Meshes	4	7	8	9	15	17	19
MG Levels	4	7	7	7	8	9	10
Finest Octant's Level	8	14	14	16	18	19	21

TABLE 4.5

Isogranular scalability for solving the variable-coefficient Poisson problem (4.2) *on a set of octrees with a grain size (on the finest multigrid level) of* $25K$ *elements (approximate) per processor* (n_p) *generated using a log-normal distribution 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 table. All timings are reported in seconds.*

CPUs	1	4	16	64	256	1024
Coarsening	0.015	0.036	0.18	0.43	0.58	0.76
Balancing	0.03	0.16	0.4	0.72	0.89	5.99
Meshing	0.28	0.59	0.76	1.26	2.73	6.13
R-setup	0.059	0.092	0.08	0.102	0.12	0.14
Total Setup	**0.95**	**0.92**	**1.33**	**3.07**	**5.15**	**14.13**
LU	0.55	0.56	0.19	0.07	0.75	0.96
R + P	0.24	0.58	0.57	1.14	0.99	1.32
Scatter	0	0	0.08	0.46	0.88	1.39
FE MatVecs	2.05	3.8	3.47	5.42	4.45	5.52
Total Solve	**3.09**	**4.96**	**4.38**	**6.48**	**6.69**	**8.48**
CG Its.	5	5	5	7	6	7
Meshes	3	4	6	11	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

speedups 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 grows with the number of processors. This is another reason why we don't observe ideal speedups for the setup phase. The speedups for the

TABLE 4.6

Fixed-size scalability for solving the variable-coefficient Poisson problem (4.2) on an octree with 31.9M elements generated from a Gaussian distribution of points. Eight multigrid levels were used. Five 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 grid, were required. All timings are reported in seconds.

CPUs	32	64	128	256	512	1024
Coarsening	9.02	5.81	4.08	2.73	1.85	1.44
Balancing	15.02	9.03	5.91	3.83	2.43	1.73
Meshing	30.69	24.81	9.25	7.99	5.94	4.15
R-setup	3.64	1.97	0.94	0.56	0.3	0.19
Total Setup	**51.14**	**37.52**	**17.32**	**13.89**	**10.39**	**7.82**
LU	1.82	2.08	1.59	1.70	1.71	1.77
R + P	24.59	12.06	7.30	4.18	2.11	1.35
Scatter	0.25	1.62	0.61	0.89	1.45	1.46
FE MatVecs	159.0	77.94	41.28	25.11	10.93	5.99
Total Solve	**181.9**	**91.59**	**48.94**	**31.23**	**15.28**	**9.98**
Meshes	10	11	12	14	15	15

TABLE 4.7

Fixed-size scalability for solving the variable-coefficient Poisson problem (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. Eight multigrid levels were used. Five iterations were required to reduce the 2-norm of the residual by a factor of 10^{-10}. All timings are reported in seconds.

CPUs	32	64	128	256	512	1024
Coarsening	5.9	4.57	2.78	1.67	1.12	0.84
Balancing	9.9	6.52	4.12	2.51	1.75	1.70
Meshing	20.17	14.28	6.61	5.39	6.4	6.17
R-setup	2.29	1.47	0.64	0.34	0.25	0.14
Total Setup	**33.51**	**24.36**	**12.59**	**9.03**	**10.36**	**9.71**
LU	0.59	1.87	1.3	0.58	0.95	0.84
R + P	13.73	9.42	4.17	2.56	2.44	1.34
Scatter	0.2	0.57	0.37	0.69	1.51	1.31
FE MatVecs	99.76	63.01	27.35	14.45	12.44	5.88
Total Solve	**113.77**	**73.86**	**32.48**	**17.41**	**16.62**	**9.1**
Meshes	10	11	12	14	15	15

solve phase, although not ideal, seem to be quite good. Poor load balancing, which affects isogranular scalability on large processor counts, seems to be another factor that affects the speedups for the setup and solve phases in the fixed-size scalability experiments.

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

TABLE 4.8

Fixed-size scalability for solving the variable-coefficient Poisson problem (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. Eight multigrid levels were used. Five iterations were required to reduce the 2-norm of the residual by a factor of 10^{-10}. All timings are reported in seconds.

CPUs	32	64	128	256	512	1024
Coarsening	2.75	2.11	0.99	0.59	0.41	0.35
Balancing	4.56	2.95	1.36	0.91	0.96	1.09
Meshing	8.46	4.95	2.68	2.68	2.61	2.59
R-setup	0.66	0.35	0.21	0.16	0.18	0.087
Total Setup	**15.54**	**9.59**	**5.52**	**4.55**	**5.21**	**7.43**
LU	1.08	0.89	0.82	0.17	0.69	0.79
R + P	4.57	2.89	1.68	0.95	0.81	0.72
Scatter	0.32	0.58	0.46	0.84	1.45	1.44
FE MatVecs	28.69	14.74	8.68	4.36	3.35	2.47
Total Solve	**35.61**	**18.49**	**11.45**	**5.65**	**5.73**	**5.12**
Meshes	11	12	13	15	15	15

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

- Finally, we have demonstrated that our implementation works well even on
  problems with variable coefficients.

There are two important extensions for the present work: higher-order discretizations and integration with domain-decomposition methods such as the hierarchical hybrid grids scheme described in [11]. 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 responsible for only a part of the entire domain.

Appendix A. Proof showing that A_k is a symmetric positive operator with respect to $(\cdot, \cdot)_k$. 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 [37]. Hence, A_k has a unique Hilbert-adjoint operator; in fact, as (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 (A.1).

$$(A_k v, v)_k = a(v, v) > 0 \quad \forall v \neq 0 \in V_k,$$
(A.1) $$(A_k w, v)_k = a(v, w) = (A_k v, w)_k = (w, A_k v)_k \quad \forall v, w \in V_k.$$

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.

$$(B.1) \qquad v = \sum_{n=1}^{\#(V_{k-1})} v_{n,k-1} \phi_n^{k-1} = \sum_{m=1}^{\#(V_k)} v_{m,k} \phi_m^k.$$

In (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

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

In (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

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

We can view the prolongation operator as a MatVec with the input vector as the coarse-grid nodal values (coefficients 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 (B.4).

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

An equivalent formulation is to satisfy (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 (B.5).

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

where

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

Since the two formulations are equivalent, we have

$$(B.7) \qquad \mathbf{P_1} = \mathbf{P_2}.$$

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

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

Since A_k is a symmetric positive operator with respect to $(\cdot,\cdot)_k$, the solution u_k to (2.4) satisfies

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

This is simply the Ritz finite element method formulation. In the multigrid scheme, we want to find

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

Here P is the prolongation operator defined in section 2.2.

$$F^k(v_k + Pw_{k-1}) = \frac{1}{2}((A_k v_k + A_k Pw_{k-1}), (v_k + Pw_{k-1}))_k - (f_k, v_k + Pw_{k-1})_k$$

$$= \frac{1}{2}(A_k v_k, v_k)_k + \frac{1}{2}(A_k Pw_{k-1}, v_k)_k + \frac{1}{2}(A_k v_k, Pw_{k-1})_k$$

$$+ \frac{1}{2}(A_k Pw_{k-1}, Pw_{k-1})_k - (f_k, v_k)_k$$

$$- \frac{1}{2}(f_k, Pw_{k-1})_k - \frac{1}{2}(f_k, Pw_{k-1})_k$$

$$= F^k(v_k) + \frac{1}{2}(A_k Pw_{k-1}, v_k)_k + \frac{1}{2}(A_k v_k - f_k), Pw_{k-1})_k$$

(C.4)
$$- \frac{1}{2}(f_k, Pw_{k-1})_k + \frac{1}{2}(P^* A_k Pw_{k-1}, w_{k-1})_{k-1}.$$

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

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

Hence, we have

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

with F_G^{k-1} defined by

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

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

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

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

(C.9)
$$(A_{k-1}^G u, v)_{k-1} = (A_k Pu, Pv)_k = (Pu, A_k Pv)_k = (u, A_{k-1}^G v)_{k-1} \quad \forall u, v \in V_{k-1},$$

$$(A_{k-1}^G u, u)_{k-1} = (A_k Pu, Pu)_k \quad \forall u \in V_{k-1},$$

$$\forall u \in V_{k-1}, \quad \exists\, w_u \in V_k \mid Pu = w_u$$

(C.10)
$$\Rightarrow (A_{k-1}^G u, u)_{k-1} = (A_k w_u, w_u)_k \geq 0 \quad \forall u \in V_{k-1}.$$

Hence, the solution v_{k-1} to (2.11) satisfies (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,

(D.1) $$w = \sum_{m=1}^{\#(V_k)} w_m \phi_m^k \quad \text{and} \quad v = \sum_{n=1}^{\#(V_{k-1})} v_n \phi_n^{k-1}.$$

Now, let

(D.2) $$R\phi_m^k = \sum_{l=1}^{\#(V_{k-1})} \mathbf{R}(l, m)\phi_l^{k-1} \quad \forall m = 1, 2, \dots, \#(V_k).$$

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

(D.3)
$$(R\phi_m^k, \phi_n^{k-1})_{k-1} = \sum_{l=1}^{\#(V_{k-1})} \mathbf{R}(l,m)(\phi_l^{k-1}, \phi_n^{k-1})_{k-1} = (\phi_m^k, \phi_n^{k-1})_k$$

$$\forall m = 1, 2, \ldots, \#(V_k) \quad \text{and} \quad \forall n = 1, 2, \ldots, \#(V_{k-1}).$$

Thus,

(D.4)
$$\boxed{\mathbf{R} = (\mathbf{M_{k-1}^{k-1}})^{-1}\mathbf{M_k^{k-1}},}$$

where

(D.5)
$$\mathbf{M_k^{k-1}}(i,j) = (\phi_i^{k-1}, \phi_j^k)_k = \mathbf{M_{k-1}^k}(j,i).$$

Appendix E. An equivalent formulation for the multigrid scheme. The coarse-grid operator defined in (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 (2.3). This operator can be implemented efficiently using a matrix-free scheme. Using (2.6), (B.5), (2.12), and (2.14), we have

(E.1)
$$\mathbf{A_{k-1}^G} = (\mathbf{M_{k-1}^{k-1}})^{-1}\tilde{\mathbf{A}}_{k-1}^G,$$
$$\tilde{\mathbf{A}}_{k-1}^G = \mathbf{M_k^{k-1}}(\mathbf{M_k^k})^{-1}\tilde{\mathbf{A}}_k(\mathbf{M_k^k})^{-1}\mathbf{M_{k-1}^k}.$$

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:

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

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

(E.3)
$$(\phi_l^k, \phi_j^{k-1})_k = \sum_{i=1}^{\#(V_k)} \mathbf{c}(i,j)(\phi_l^k, \phi_i^k)_k \; \forall j = 1, 2, \ldots, \#(V_{k-1}) \text{ and } \forall l = 1, 2, \ldots, \#(V_k).$$

This leads to

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

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

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

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

(E.6)
$$\tilde{A}_k u_k = \tilde{f}_k,$$

and we solve the following corresponding coarse-grid problem

$$(E.7) \qquad \tilde{\mathbf{A}}_{\mathbf{k-1}}\mathbf{e}_{\mathbf{k-1}} = \mathbf{M}_{\mathbf{k-1}}^{\mathbf{k-1}}\mathbf{f}_{\mathbf{k-1}}^{\mathbf{G}} = \tilde{\mathbf{R}}\mathbf{r}_{\mathbf{k}} = \mathbf{r}_{\mathbf{k-1}}$$

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

$$(E.8) \qquad \tilde{\mathbf{R}} = \mathbf{M}_{\mathbf{k-1}}^{\mathbf{k-1}}\mathbf{R}(\mathbf{M}_{\mathbf{k}}^{\mathbf{k}})^{-1}.$$

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

$$(E.9) \qquad \boxed{\tilde{\mathbf{R}} = \mathbf{P_2}^T.}$$

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

Appendix F. Pseudocodes for some components. Here we state several algorithms (Algorithms 2–7) that are required to efficiently implement the overall method.

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. $c \leftarrow (4k + 2j + i)$

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. $C \leftarrow \emptyset$
2. $\mathcal{I}_1 \leftarrow 0$
3.   **while** $(\mathcal{I}_1 < \mathbf{len}(F))$
4.   Find $\mathcal{I}_2$ such that $\mathbf{Child-Number}(F[\mathcal{I}_2]) = 0$ and
             $\mathbf{Child-Number}(F[k]) \neq 0 \quad \forall \quad \mathcal{I}_1 < k < \mathcal{I}_2$.
5.   **if** no such $\mathcal{I}_2$ exists
6. $\mathcal{I}_2 \leftarrow \mathbf{len}(F)$
7.   **end if**
8.   **if** $\mathcal{I}_2 \geq (\mathcal{I}_1 + 8)$
9. $C.\mathbf{push\_back}(\mathbf{Parent}(F[\mathcal{I}_1]))$
10.   **if** $\mathcal{I}_2 > (\mathcal{I}_1 + 8)$
11. $C.\mathbf{push\_back}(F[\mathcal{I}_1 + 8], F[\mathcal{I}_1 + 9], \ldots, F[\mathcal{I}_2 - 1])$
12.   **end if**
13.   **else**
14. $C.\mathbf{push\_back}(F[\mathcal{I}_1], F[\mathcal{I}_1 + 1], \ldots, F[\mathcal{I}_2 - 1])$
15.   **end if**
16. $\mathcal{I}_1 \leftarrow \mathcal{I}_2$
17. **end while**

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 a nonblocking MPI send.
11. **end if**
12. **if** P is not the last processor
13.   Send M_l to the next processor (P+1)
         using a nonblocking 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 $\left\{F[\mathcal{I}_f], F[\mathcal{I}_f + 1], \ldots, F[\mathcal{I}_l - 1]\right\}$
         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**

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

1.   **if** $\mathcal{I}_p \geq 0$ and $M_f \geq 0$
2.   **if** $(\mathcal{I}_p + \mathcal{I}_f) \geq 8$
3. $\mathcal{I}_c \leftarrow \max(0, (8 - \mathcal{I}_p))$
4. $C.\mathbf{push\_front}(F[\mathcal{I}_c], F[\mathcal{I}_c + 1], \ldots, F[\mathcal{I}_f - 1])$
5.   **else**
6. $C.\mathbf{push\_front}(F[0], F[1], \ldots, F[\mathcal{I}_f - 1])$
7.   **end if**
8.   **else**
9.   **if** $M_f < 0$
10.   **if** $\mathcal{I}_p < 0$ or $\mathcal{I}_p \geq 8$
11. $C \leftarrow F$
12.   **else**
13. $\mathcal{I}_c \leftarrow (8 - \mathcal{I}_p)$
14. $C.\mathbf{push\_front}(F[\mathcal{I}_c], F[\mathcal{I}_c + 1], \ldots, F[\mathcal{I}_f - 1])$
15.   **end if**
16.   **else**
17. $C.\mathbf{push\_front}(F[0], F[1], \ldots, F[\mathcal{I}_f - 1])$
18.   **end if**
19. **end if**

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

---

1.   **if** $\mathcal{I}_n \geq 0$ **and** $M_l \geq 0$
2.   **if** $(\mathcal{I}_n + M_l) \geq 8$
3. C.**push_back**(**Parent**($F[\mathcal{I}_l]$))
4.   **if** $M_l > 8$
5. C.**push_back**($F[\mathcal{I}_l + 8], F[\mathcal{I}_l + 9], \ldots, F[\textbf{len}(F) - 1]$)
6.   **end if**
7.   **else**
8. C.**push_back**($F[\mathcal{I}_l], F[\mathcal{I}_l + 1], \ldots, F[\textbf{len}(F) - 1]$)
9.   **end if**
10. **else**
11.   **if** $M_l \geq 0$
12. C.**push_back**(**Parent**($F[\mathcal{I}_l]$))
13.   **if** $M_l > 8$
14. C.**push_back**($F[\mathcal{I}_l + 8], F[\mathcal{I}_l + 9], \ldots, F[\textbf{len}(F) - 1]$)
15.   **end if**
16.   **end if**
17. **end if**

ALGORITHM 7.   PARALLEL RESTRICTION MATVEC
(AS EXECUTED BY PROCESSOR P)

Input: Fine vector (F), masks (M), precomputed stencils (R_1) and (R_2), fine octree (O_f), coarse octree (O_c).
Output: Coarse vector (C).
Note: For simplicity, we do not overlap communication with computation in the pseudocode. In the actual implementation, we overlap communication with computation as described in section 3.4.4.

1.   Exchange ghost values for F and M with other processors.
2. $C \leftarrow 0$.
3.   **for each** $o^c \in O_c$
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 $o^f \in O_f$ is found such that
         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 ith 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 jth vertex of o^c.
13.   If V_c is hanging, use the corresponding
                    nonhanging vertex instead.
14.   **if** the jth 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 received from other processors
     to the local copy of C.

REFERENCES

[1] M. F. ADAMS, H. H. BAYRAKTAR, T. M. KEAVENY, AND P. PAPADOPOULOS, *Ultrascalable implicit finite element analyses in solid mechanics with over a half a billion degrees of freedom*, in Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, ACM/IEEE, Pittsburgh, PA, ACM, New York, 2004.

[2] M. ADAMS AND J. W. DEMMEL, *Parallel multigrid solver for 3d unstructured finite element problems*, in Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, Portland, OR, ACM Press, New York, 1999.

[3] V. AKCELIK, J. BIELAK, G. BIROS, I. EPANOMERITAKIS, A. FERNANDEZ, O. GHATTAS, E. J. KIM, J. LOPEZ, D. R. O'HALLARON, T. TU, AND J. URBANIC, *High resolution forward and inverse earthquake modeling on terascale computers*, in Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, Phoenix, AZ, ACM, New York, 2003.

[4] D. AMBROSI AND F. MOLLICA, *On the mechanics of a growing tumor*, Internat. J. Engrg. Sci., 40 (2002), pp. 1297–1316.

[5] W. K. ANDERSON, W. D. GROPP, D. K. KAUSHIK, D. E. KEYES, AND B. F. SMITH, *Achieving high sustained performance in an unstructured mesh CFD application*, in Proceedings of the 1999 ACM/IEEE Conference on Supercomputing, 1999, ACM, New York, pp. 69–69.

[6] S. BALAY, K. BUSCHELMAN, W. D. GROPP, D. KAUSHIK, M. G. KNEPLEY, L. C. McINNES, B. F. SMITH, AND H. ZHANG, *PETSc home page*, http://www.mcs.anl.gov/petsc, 2001.

[7] R. E. BANK AND T. DUPONT, *An optimal order process for solving finite element equations*, Math. Comp., 36 (1981), pp. 35–51.

[8] P. BASTIAN, W. HACKBUSCH, AND G. WITTUM, *Additive and multiplicative multigrid: A comparison*, Computing, 60 (1998), pp. 345–364.

[9] R. BECKER, M. BRAACK, AND T. RICHTER, *Parallel multigrid on locally refined meshes*, in Reactive Flows, Diffusion and Transport, Springer, Berlin, 2007, pp. 77–92.

[10] R. BECKER AND M. BRAACK, *Multigrid techniques for finite elements on locally refined meshes*, Numer. Linear Algebra Appl., 7 (2000), pp. 363–379.

[11] B. BERGEN, F. HULSEMANN, AND U. RUDE, *Is 1.7×10^{10} unknowns the largest finite element system that can be solved today?*, in Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, IEEE, Washington, DC, 2005.

[12] M. W. BERN, D. EPPSTEIN, AND S.-H. TENG, *Parallel construction of quadtrees and quality triangulations*, Internat. J. Comput. Geom. Appl., 9 (1999), pp. 517–532.

[13] 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, 1997.

[14] D. BRAESS AND W. HACKBUSCH, *A new convergence proof for the multigrid method including the V-cycle*, SIAM J. Numer. Anal., 20 (1983), pp. 967–975.

[15] J. H. BRAMBLE, J. E. PASCIAK, AND J. XU, *The analysis of multigrid algorithms for nonsymmetric and indefinite elliptic problems*, Math. Comp., 51 (1988), pp. 389–414.

[16] J. H. BRAMBLE, J. E. PASCIAK, AND J. XU, *Parallel multilevel preconditioners*, Math. Comp., 55 (1990), pp. 1–22.

[17] S. C. BRENNER AND L. R. SCOTT, *The mathematical theory of finite element methods*, in Texts Appl. Math. 15, Springer, New York, 1994.

[18] W. L. BRIGGS, V. E. HENSON, AND S. F. McCORMICK, *A Multigrid Tutorial* 2nd ed., SIAM, Philadelphia, 2000.

[19] H.-J. BUNGARTZ, M. MEHL, AND T. WEINZIERL, *A parallel adaptive Cartesian PDE solver using space–filling curves*, in Parallel Processing, 12th International Euro-Par Conference, W. E. Nagel, W. V. Walter, and W. Lehner, eds., Lecture Notes in Comput. Sci. 4128, Springer, Berlin, 2006, pp. 1064–1074.

[20] P. M. CAMPBELL, K. D. DEVINE, J. E. FLAHERTY, L. G. GERVASIO, AND J. D. TERESCO, *Dynamic Octree Load Balancing Using Space-Filling Curves*, Technical report CS-03-01, Department of Computer Science, Williams College, Williamstown, MA, 2003.

[21] W. M. DEEN, *Analysis of Transport Phenomena*, Topics in Chemical Engineering, Oxford University Press, New York, 1998.

[22] J. E. DENDY, *Black box multigrid*, J. Comput. Phys., 48 (1982), pp. 366–386.

[23] R. FALGOUT, A. CLEARY, J. JONES, E. CHOW, V. HENSON, C. BALDWIN, P. BROWN, P. VASSILEVSKI, AND U. M. YANG, *Hypre home page*, http://acts.nersc.gov/hypre, 2001.

[24] R. FALGOUT, *An introduction to algebraic multigrid*, Comput. Sci. Engrg., 8 (2006), pp. 24–33.

[25] M. GEE, C. SIEFERT, J. HU, R. TUMINARO, AND M. SALA, *ML 5.0 Smoothed Aggregation User's Guide*, Technical report SAND2006-2649, Sandia National Laboratories, Albuquerque, NM, 2006.

[26] R. GLOWINSKI, T.-W. PAN, T. I. HESLA, D. D. JOSEPH, AND J. PERIAUX, *A fictitious domain method with distributed Lagrange multipliers for the numerical simulation of particulate flow*, Contemp. Math., 218 (1998), pp. 121–137.

[27] L. GORELICK, M. GALUN, AND A. BRANDT, *Shape representation and classification using the Poisson equation*, IEEE Trans. Pattern Anal. Mach. Intelligence, 28 (2006), pp. 1991–2005.

[28] M. GRIEBEL AND G. ZUMBUSCH, *Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves*, Parallel Comput., 25 (1999), pp. 827–843.

[29] D. J. GRIFFITHS, *Introduction to Electrodynamics*, Prentice-Hall, Englewood Cliffs, NJ, 1999.

[30] D. J. GRIFFITHS, *Introduction to Quantum Mechanics*, Prentice-Hall, Englewood Cliffs, NJ, 2004.

[31] W. D. GROPP, D. K. KAUSHIK, D. E. KEYES, AND B. F. SMITH, *Performance modeling and tuning of an unstructured mesh CFD application*, in Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, Washington, DC, IEEE, Dallas, TX, 2000.

[32] E. HABER AND S. HELDMANN, *An octree multigrid method for quasi-static Maxwell's equations with highly discontinuous coefficients*, J. Comput. Phys., 223 (2007), pp. 783–796.

[33] W. HACKBUSCH, *Multigrid methods and applications*, Springer Ser. Comput. Math. 4, Springer, Berlin, 1985.

[34] A. JONES AND P. JIMACK, *An adaptive multigrid tool for elliptic and parabolic systems*, Internat. J. Numer. Methods in Fluids, 47 (2005), pp. 1123–1128.

[35] M. JUNG AND U. RUDE, *Implicit extrapolation methods for variable coefficient problems*, SIAM J. Sci. Comput., 19 (1998), pp. 1109–1124.

[36] 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, San Francisco, CA, 2002.

[37] E. KREYSZIG, *Introductory Functional Analysis with Applications*, John Wiley, New York, 1989.

[38] S. LANG, *Parallel-adaptive simulation with the multigrid-based software framework UG*, Engrg. with Comput., 22 (2006), pp. 157–179.

[39] X. S. LI AND J. W. DEMMEL, *SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Trans. Math. Software, 29 (2003), pp. 110–140.

[40] D. J. MAVRIPLIS, M. J. AFTOSMIS, AND M. BERGER, *High resolution aerospace applications using the NASA Columbia Supercomputer*, in Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, IEEE, Washington, DC, 2005.

[41] M. MEHL, *Cache-optimal data-structures for hierarchical methods on adaptively refined space-partitioning grids*, in 2006 International Conference on High Performance Computing and Communications, Bangalore, India, 2006.

[42] J. MODERSITZKI, *Numerical Methods for Image Registration*, Numer. Math. Sci. Comput., Oxford University Press, New York, 2004.

[43] M. T. NEVES-PETERSEN AND S. 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 (2003), pp. 315–395.

[44] S. POPINET, *Gerris: A tree-based adaptive solver for the incompressible Euler equations in complex geometries*, J. Comput. Phys., 190 (2003), pp. 572–600.

[45] J. B. PORMANN, C. S. HENRIQUEZ, J. JOHN, A. BOARD, D. J. ROSE, D. M. HARRILD, AND A. P. HENRIQUEZ, *Computer simulations of cardiac electrophysiology*, in Proceedings of the 2000 IEEE/ACM Conference on Supercomputing, Washington, DC, IEEE, Dallas, TX, 2000.

[46] I. RAMIÈRE, P. ANGOT, AND M. BELLIARD, *A general fictitious domain method with immersed jumps and multilevel nested structured meshes*, J. Comput. Phys., 225 (2007), pp. 1347–1387.

[47] R. S. SAMPATH, S. S. ADAVANI, H. SUNDAR, I. LASHUK, AND G. BIROS, *Dendro: Parallel algorithms for multigrid and AMR methods on 2:1 balanced octrees*, in Proceedings of

the 2008 ACM/IEEE Conference on Supercomputing, Austin, TX, IEEE, Piscataway, NJ, 2008, pp. 1–12.

[48] R. Sampath, H. Sundar, S. S. Adavani, I. Lashuk, and G. Biros, `Dendro`: *A Parallel Geometric Multigrid Library for Finite Elements on Octree Meshes*, http://www.cc.gatech.edu/csela/dendro, 2008.

[49] R. Sampath, H. Sundar, S. S. Adavani, I. Lashuk, and G. Biros, `Dendro` *Users Manual*, Technical report, Georgia Institute of Technology, Atlanta, GA, 2008.

[50] Y. Shapira, *Multigrid for locally refined meshes*, SIAM J. Sci. Comput., 21 (1999), pp. 1168–1190.

[51] S. P. Spekreijse, *Elliptic grid generation based on Laplace equations and algebraic transformations*, J. Comput. Phys., 118 (1995), pp. 38–61.

[52] H. Sundar, R. S. Sampath, S. S. Adavani, C. Davatzikos, and G. Biros, *Low-constant parallel algorithms for finite element simulations using linear octrees*, in Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, Reno, NV, ACM Press, New York, 2007.

[53] H. Sundar, R. S. Sampath, and G. Biros, *Bottom-up construction and* 2:1 *balance refinement of linear octrees in parallel*, SIAM J. Sci. Comput., 30 (2008), pp. 2675–2708.

[54] TACC, *Ranger's system architecture.* http://www.tacc.utexas.edu.

[55] O. Tatebe and Y. Oyanagi, *Efficient implementation of the multigrid preconditioned conjugate gradient method on distributed memory machines*, in Proceedings of the 1994 ACM/IEEE Conference on Supercomputing, Washington, DC, ACM, New York, 1994, pp. 194–203.

[56] H. Tropf and H. Herzog, *Multidimensional range search in dynamically balanced trees*, Angew Inform., 2 (1981), pp. 71–77.

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

[58] T. Tu, D. R. O'Hallaron, and O. Ghattas, *Scalable parallel octree meshing for terascale applications*, in Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, Seattle, WA, IEEE, Washington, DC, 2005.

[59] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'Hallaron, *From mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing*, in Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, Tampa, FL, ACM Press, New York, 2006.

[60] B. S. White, S. A. McKee, B. R. de Supinski, B. Miller, D. Quinlan, and M. Schulz, *Improving the computational intensity of unstructured mesh applications*, in Proceedings of the 19th Annual International Conference on Supercomputing, Dresden, Germany, ACM Press, New York, 2005, pp. 341–350.

[61] H. Yserentant, *On the convergence of multi-level methods for strongly nonuniform families of grids and any number of smoothing steps per level*, Computing, 30 (1983), pp. 305–313.

[62] H. Yserentant, *The convergence of multilevel methods for solving finite-element equations in the presence of singularities*, Math. Comp., 47 (1986), pp. 399–409.

[63] S. Zhang, *Optimal-order nonnested multigrid methods for solving finite element equations.* I. *On quasi-uniform meshes*, Math. Comp., 55 (1990), pp. 23–36.

[64] S. Zhang, *Optimal-order nonnested multigrid methods for solving finite element equations.* II. *On nonquasiuniform meshes*, Math. Comp., 55 (1990), pp. 439–450.