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

Rahul S. Sampath, Santi S. Adavani, Hari Sundar, Ilya Lashuk, and George Biros
University of Pennsylvania

Abstract

In this article, we present `Dendro`, a suite of parallel algorithms for the discretization and solution of partial differential equations that require discretization of second-order elliptic operators. `Dendro` uses trilinear finite element discretizations constructed using octrees. `Dendro`, which is built on top of PETSc (Argonne National Laboratories), comprises four main modules: *a bottom-up octree generation and 2:1 balancing module, a meshing module, a geometric multiplicative multigrid module, and a module for adaptive mesh refinement (AMR)*. The first two components constitute prior work that has been published elsewere. Here, we focus on the multigrid module and AMR modules. The key features of `Dendro` are coarsening/refinement, inter-octree transfers of scalar and vector fields, and parallel partition of multilevel octree forests. We describe an algorithm for constructing the coarser multigrid levels starting with an arbitrary 2:1 balanced fine grid octree discretization. We also describe matrix-free implementations for the discretized finite element operators and the intergrid transfer operations. The current implementation of `Dendro` is most appropriate for problems with smooth variable coefficients.

We present scalability results for a Poisson problem, a linear elastostatics problem, and for a time-dependent heat equation. We use the first two equations to illustrate the effectiveness of the multigrid solver. We use the third equation to illustrate the performance of the AMR components. We present results on up to 4096 CPUs on the Cray XT3 ("BigBen") at the Pittsburgh Supercomputing Center (PSC) and the Intel 64 system ("Abe") at the National Center for Supercomputing Applications (NCSA).

1 Introduction

Second-order elliptic operators, like the Laplacian operator, are ubiquitous in computational science and engineering. They effectively model diffusive transport, viscous dissipation in fluids, and stress-strain relationships in solid mechanics. The need for high-resolution simulation of such PDEs requires scalable discretization and solution schemes.

Octree meshes strike a balance between the simplicity of structured meshes and the adaptivity of generic unstructured meshes [24, 25]. On multithousand-CPU platforms, the resulting octree-based discretized operators must be "inverted" using iterative solvers. Iterative solvers for finite element based discretizations must address the ill-conditioning of second order operators, which deteriorates with increasing problem size. Multigrid algorithms (geometric and algebraic) provide a powerful mathematical framework that allows the construction of solvers with optimal algorithmic complexity [23]. Numerous sequential and parallel implementations for both geometric and algebraic multigrid methods are available.

Related work on parallel multigrid. Excellent surveys on parallel algorithms for multigrid can be found in [9] and [15].[1] Here, we give a brief (and incomplete!) overview of distributed memory message-passing based parallel multigrid algorithms. The two basic approaches are algebraic multigrid and geometric multigrid. The advantages of algebraic multigrid is that it can be used in a black-box fashion with unstructured grids and a great variety of operators (e.g., operators with discontinuous coefficients). The disadvantage of existing implementations is the relatively high setup costs. The main advantage of geometric multigrid approaches is that it is easier to devise coarsening and intergrid transfers with low overhead. Its disadvantage is that it is can not be used in a black-box fashion as it depends on both the PDE and the scheme used for its discretization.

Our interest is in developing multigrid methods for highly non-uniform meshes. Currently, the most scalable methods are based on graph-based methods for coarsening, namely maximal-matchings. Examples of scalable codes that use such graph-based coarsening include [1] and [18]. Another powerful code for algebraic multigrid is Hypre [11, 10]. Hypre has been used extensively to solve problems with a variety of operators on structured and unstructured grids. 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 us to design geometric multigrid for octree-based data structures.

Many geometric multigrid algorithms for nonuniform discretizations have been proposed in the past. In [7], a sequential geometric multigrid algorithm was proposed for finite elements on non-nested unstructured meshes, but with intergrid transfer operations that is difficult to parallelize. A sequential multigrid scheme for finite element simulations on quadtree meshes was described in [16]. In addition to the 2:1 balance constraint, a number of 'safety layers' of octants were added at each level to support intergrid transfer operations. Projections were required at each level to preserve the continuity of the solution, which is otherwise not guaranteed using their non-conforming discretizations. In [3], the authors describe a Poisson solver with excellent scalability. Strictly speaking, it is not a bona fide multigrid solver as there is no iteration between the multiple levels. Instead, it is based on local independent solves in nested grids followed by global corrections to restore smoothness. It similar to block structured grids and its extension to arbitrarily graded grids is not immediately obvious. One of the largest calculations was reported in [5], using conforming discretizations and geometric multigrid solvers on semi-structured meshes. This approach is highly scalable for nearly structured meshes and for constant coefficient PDEs. However, the scheme is not efficient when deviating from that structure.

Multigrid methods on octrees have been proposed and developed for sequential and modestly parallel adaptive finite element implementations [4, 13]. A characteristic of octree meshes is that they contain 'hanging nodes'. In [21], we presented a strategy to tackle these hanging nodes and build conforming, trilinear finite element discretizations on these meshes. This algorithm scaled up to four billion octants on 4096 CPUs on a Cray XT3 at the Pittsburgh Supercomputing Center. We also showed that the cost of applying the Laplacian using this framework is comparable to that of applying it using a direct indexing regular grid discretization with the same number of elements. ***To our knowledge, there is***

[1] We do not attempt to review the extensive literature on methods for adaptive mesh refinement. Our work is restricted on simple intergrid transfers for different octrees.

no work on parallel, octree-based, geometric multigrid solvers for finite element discretizations. Here, we build on our previous work and present a novel parallel geometric multigrid scheme.

Contributions. Our goal is to minimize storage requirements, obtain low setup costs, and achieve end-to-end[2] parallel scalability:

- We propose a parallel global **coarsening algorithm** to construct a hierarchy of nested 2:1 balanced octrees and their corresponding meshes starting with an arbitrary 2:1 balanced fine octree. We do not impose any restrictions on the number of meshes in this sequence or the size of the coarsest mesh. The process of constructing coarser meshes from a fine mesh is harder than iterative global refinements of a coarse mesh because we must ensure that the coarser grids satisfy the 2:1 balanced constraint as well; while this is automatically satisfied for the case of global refinements, it is not true with global coarsening. However, this bottom-up approach is more natural for certain applications (e.g., evolution PDEs), in which only the fine mesh is available.

- Transferring information between successive multigrid levels is a challenging task on unstructured meshes and especially so in parallel since the coarse and fine grids need not be aligned and near neighbor searches are required. Here, we describe a matrix-free algorithm for **intergrid transfer operators** that uses the special properties of an octree data structure.

- We have integrated the above mentioned components in a parallel matrix-free implementation of a geometric multiplicative multigrid method for finite elements on octree meshes. Our MPI-based implementation, `Dendro` has scaled to billions of elements on thousands of CPUs even for problems with large contrasts in the material properties. `Dendro` ***is an open source code that can be download from*** [20]. `Dendro` is tightly integrated with PETSc [2].

In the following, we will use the term "*MatVec*" to denote a matrix-vector multiplication, we well use "*octants*" or "*nodes*" to refer to octree nodes and "*vertices*" to refer to element vertices. We will use "*CPUs*" to refer to message passing processes. We will use "*hanging nodes/vertices*" to refer to octree nodes with special properties, which we explain later. There is a one-to-one correspondence between nodes, elements, and vertices.

Limitations of `Dendro`. Here, we briefly summarize the limitations of the proposed methodology. More details can be found in [19]. **(1)** The restriction and prolongation operators and the coarse grid approximations are not efficient for problems with discontinuous coefficients. **(2)** The method does not work for strongly indefinite elliptic operators (e.g., high-frequency Helmholtz problems). **(3)** Currently `Dendro` supports only Dirichlet and Neumann conditions. **(4)** `Dendro` is limited to second-order accurate discretization of elliptic operators on the unit cube. Problems with complex geometries are not directly supported although `Dendro` can be used with penalty approaches to allow solution of such problems. **(5)** Only the intergrid transfers have been implemented in the AMR module (no error estimators). **(6)** Load balancing is not

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

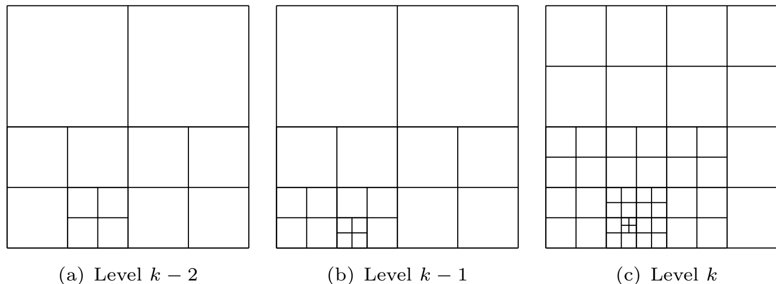


Figure 1: *Quadtree meshes for three successive multigrid levels.*

addressed fully. There are issues due to the 2:1 balance constraint and coarsening that can prevent perfect speed ups. **(7)** The interlevel partitioning scheme does not use neither load information across multiple levels. *We are currently working to address items (5) and (6) and we are optimistic that improved results on up to 9K CPUs on NCSA's Intel cluster.*

Organization of the paper. In Section 2, we describe a matrix-free implementation for the multigrid method. We describe our framework for handling 'hanging' nodes and how we use it to implement the MatVecs for the finite element matrices and the restriction/prolongation matrices. In Section 3, we present the results from fixed-size and iso-granular scalability experiments. We also compare our implementation with 'BoomerAMG' [14], an algebraic multigrid implementation available in Hypre. In Section 4, we present the conclusions from this study and discuss ongoing and future work.

2 Parallel Geometric Multigrid

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

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

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

The `Dendro` interface. Let us describe the `Dendro` interface for (1). (The current implementation of `Dendro` supports more general scalar equations, vector equations, and time-dependent parabolic, and hyperbolic equations.) The input is μ, f, boundary conditions, and a list of target points in which we want to evaluate the solution. The output is the solution u and, optionally, its gradient at the target points. Here, for simplicity, we describe only the representation of

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

μ: it is approximated as the sum of a constant plus a function that is defined by specifying its value on a set of points. f is represented in a similar manner. We assume that the lists of points that are used to specify μ and f are arbitrarily distributed across CPUs. `Dendro` uses these points to define the fine-level octree and subsequently the overall multigrid hierarchy. Next, we give details about the main algorithmic components of this procedure.

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

Octree Forest construction	Bottom-up Coarsening	Top-Down Meshing
1. Bottom-up construct fine octree 2. Partition fine octree 3. Balance fine octree 4. Bottom-up coarsen octree 5. Top-down meshing	1. Replace leafs with their parent 2. Balance and Morton-order partition 3. Repeat	1. Mesh and partition coarse grid 2. Prolong partitioning to next grid 3. Check load balancing and repartition 4. Construct prolongation operator

Table 1: *Summary of main algorithmic components for the multigrid construction used in* Dendro *[19].*

We use the standard Jacobi smoothing. Due to space limitations, we do not discuss more advanced smoothers. Let us just say that any read-only-ghost type smoother can be implemented with no additional communication. For the exact solve we use SuperLU [17]. The operators A_k are defined using standard FEM discretization (so in the general case, the coarse grid operators do not satisfy the so-called Galerkin condition). We say more about this and the coarse-grid representation in the next section.

To construct the solver we start with user input. Given the points for μ and f, we merge them and use them to bottom-up construct the fine-level octree using the methods we described in [21]. The main challenge is the construction of the hierarchy of meshes and the corresponding prolongation and restriction operators. Therefore, the basic steps in constructing the multigrid step are the following:

1. Given input points, construct and partition octree at the fine level.
2. Given the total number of levels, construct and partition the coarser octrees.
3. Mesh the octrees.
4. Construct restriction and prolongation operators.

The basic algorithms that we use to implement these steps are outlined in Table 1. In the following, we briefly summarize the octree construction and then we explain the rest of the algorithm.

Smoothing and coarse grid operator. One of the problems with geometric multigrid methods is that their performance deteriorates with increasing contrast in material properties. In matrix-free geometric multigrid implementations the coarse grid operator is not constructed using the Galerkin condition, instead the coarser grid discretization of problem is used as the coarse grid operator. We can easily show [19] that these two formulations are equivalent provided the same bilinear form, $a(u, v)$, is used both on the coarse and fine levels. This poses no difficulty for constant coefficient problems or problems in which the material property is described in a closed form. However, sometimes the material property is just defined on each fine grid element. Hence, the bilinear form actually depends on the discretization used. If the coarser grids must use the same bilinear form, the coarser grid MatVecs must be performed by looping over the underlying finest grid elements, using the material property defined on each fine grid element. This would make the coarse grid MatVecs quite expensive. A cheaper alternative would be to define the material properties for the coarser grid elements as the average of those for the underlying fine grid elements. This process amounts to using a different bilinear form for each multigrid level and hence is a clear deviation from the theory. Hence, the convergence of the stand-alone multigrid solver deteriorates with increasing contrast in material properties. However, the multigrid scheme is known to be a good preconditioner to the Conjugate Gradient method [23]. We have conducted numerical experiments that demonstrate this for the Poisson problem.

2.1 Octree construction

Each octree node has a maximum of eight children. A node with no children is called a *leaf* and a node with one or more children is called an *interior node*. Complete octrees are trees in which every interior node has exactly eight children. The only node with no parent is the *root* and all other nodes have exactly one parent. Nodes that have the same parent are called *siblings*.

At each level, we use a linear octree representation using a Morton encoding to represent the position and level of the octant of the tree locational code to identify the octants [8]. In order to construct a Morton encoding, the maximum permissible depth, D_{max}, of the tree is specified a priori. Next, we assume that the domain is represented by an imaginary uniform grid of $2^{D_{max}}$ indivisible cells in each dimension. Each cell is identified by an integer triplet representing its x, y and z coordinates, respectively. 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 it's lower left corner facing the reader. The Morton encoding for any octant is then derived by interleaving the binary representations (D_{max} bits each) of the three coordinates of the octant's anchor, and then appending the binary representation $((\lfloor(\log_2 D_{max})\rfloor + 1)$ bits) of the octant's level to this sequence of bits [6].

In many applications involving octrees, it is desirable to impose a restriction on the relative sizes of adjacent octants, also known as the "*2:1 balance constraint*" (not to be confused with load balancing) that requires no leave at level l shares a corner, edge, or face with another leaf at a level greater than $l + 1$.

Nodes that exist at the center of a face of an octant are called *face-hanging* nodes and those that are located at the mid-

point of an edge are called *edge-hanging* nodes. The 2:1 balance constraint ensures that there is at most one hanging node on any edge or face. In [22], we presented a parallel algorithm for linear octree construction and 2:1 balancing and which is a key component of `Dendro`. This balancing algorithm has an $\mathcal{O}(N \log N)$ work complexity and an $\mathcal{O}(\frac{N}{n_p} \log \frac{N}{n_p} + n_p \log n_p)$ parallel time complexity, with N being the problem size and n_p the number of CPUs.

To partition the tree for balancing end meshing, we introduced "*Block Partition*", a heuristic algorithm for a single-level octree. We first partition the octree using Morton-ordering and then we build a smaller complete linear octree whose leaves we term *blocks*. This auxiliary octree encapsulates and compresses the local spatial distribution of the input octants. Moreover, every octant in the input is either a block in itself or is a descendant of some block. We then compute the number of original octants that lie within each block and assign this number as the block's weight. We use a weighted Morton partitioning algorithm to partition these blocks. Finally, the computed partition is projected onto the input octants. The result of this is that the octants in the input and the blocks containing them are sent to the same CPU.

2.2 Global coarsening

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

We construct the k-th octree from the $k+1$-th octree by replacing every set of eight siblings by their parent. This is an operation with $\mathcal{O}(N)$ work complexity, where N is the number of leaves in the $k+1$-th octree. It is easy to parallelize and has an $\mathcal{O}(\frac{N}{n_p})$ parallel time complexity, where n_p is the number of CPUs.[4] The main parallel operations are two circular shifts; one clockwise and another anti-clockwise.

However, the operation described above may produce 4:1 balanced octrees[5] instead of 2:1 balanced octrees. Although there is only one level of imbalance that we need to correct, the imbalance can still affect octants that are not in its immediate vicinity. This is known as the *ripple effect*. Even with just one level of imbalance, a ripple can still propagate across many CPUs.

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

2.3 Meshing

By meshing, we refer to the construction of the octree-node to the mesh-vertex and element mappings, the partitioning and construction of ghost and local vertex lists, and the construction of the scatter/gather operators for the near-neighbor

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

[5] 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.

communications that are required during the MatVec.

In [21], we developed algorithms and efficient data structures that support trilinear conforming finite element calculations on linear octrees. We use these data structures in the present work too. The key features of the algorithm are that (1) we use a hanging-node management scheme that allows MatVecs in a single tree traversal as opposed to multiple tree traversal required by the scheme in [24], and (2) we reduce the memory overhead by storing the octree in a compressed form that requires only one byte per octant (used to store the level of the octant). The element-to-mesh vertex mappings can be compressed at a modest expense of uncompressing this on the fly while looping over the elements to perform the finite element MatVecs. The resulting shape functions on the octree meshes are trilinear, are equal to 1 at the vertex at which they are rooted, vanish at all other non-hanging vertices in the octree, and their support can spread over more than 8 elements.

Here, we extend the meshing algorithm to the case of a forest of octrees. The main difference with the single-level algorithm is that the partitionings between different levels are coupled. We give details below.

2.3.1 Multilevel meshing and the prolongation operators

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

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

To satisfy these conditions, *we first compute the partition on the coarse grid and then impose it on the finer grid*. In general, it might not be possible or desirable to use the same partition for all of the levels. For example, the coarser levels might be too sparse to be distributed across all the CPUs or using the same partition for all the levels could lead to a large load imbalance across the CPUs. Hence, we allow some levels to be partitioned differently than others[6]. When a transition in the partitions is required, we duplicate the octree in question and let one of the duplicates share the same partition

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

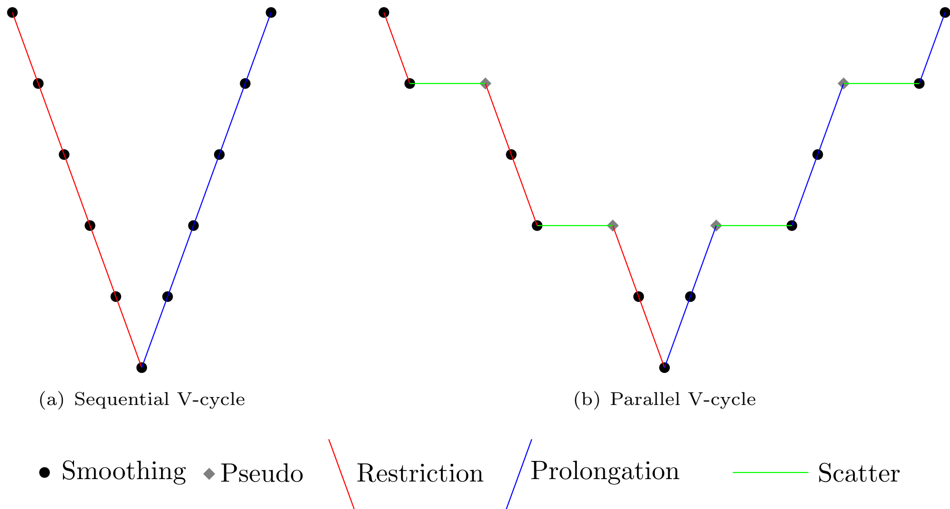


Figure 2: *(a) A V-cycle where the meshes at all levels share the same partition and (b) A V-cycle where not all meshes share the same partition. Some meshes do share the same partition and whenever the partition changes a 'pseudo' mesh is added. The 'pseudo' mesh is only used to support intergrid transfer operations and smoothing is not performed on this mesh.*

as that of its immediate finer level and let the other one share the same partition as that of its immediate coarser level. We refer to one of these duplicates as the 'pseudo' mesh (Figure 2). The 'pseudo' mesh is only used to support intergrid transfer operations and smoothing is not performed on this mesh. On these levels, the intergrid transfer operations include an additional step referred to as *Scatter*, which just involves re-distributing the values from one partition to another.

One of the challenges with the MatVec for the intergrid transfer operations is that as we loop over the octants we must also keep track of the pairs of coarse and fine grid nodes that were visited already. In order, to implement this MatVec efficiently we make use of the following observations. **(1)** Every non-hanging fine grid node is shared by at most eight fine grid elements, excluding the elements whose hanging vertices are mapped to this node. **(2)** Each of these eight fine grid elements will be visited only once within the Restriction and Prolongation MatVecs. **(3)** Since we loop over the coarse and fine 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. **(4)** The only coarse grid shape functions that do not vanish at the non-hanging fine grid node under consideration are those whose indices are stored in the vertices of each of these coarse octants. Some of these vertices may be hanging, but they will be mapped to the corresponding non-hanging vertex. So, the correct index is always stored immaterial of the hanging state of the vertex.

We pre-compute and store a mask for each fine grid node. Each of these masks is a set of eight bytes, one for each of the eight fine grid elements that surround this fine grid node. 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 node. The overhead of using this mask within the actual MatVecs is just the cost of a few bitwise operations for each fine grid octant. Algorithm 1 lists the sequence of operations performed by a CPU for the restriction MatVec. This MatVec is an operation with $\mathcal{O}(N)$ work complexity and has an $\mathcal{O}(\frac{N}{n_p})$ parallel time complexity. For simplicity, we do not overlap communication

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

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

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

Computing the 'masks' for restriction and prolongation. Each non-hanging fine grid vertex has a maximum[7] of 1758 unique locations at which a coarse grid shape function that contributes to this vertex could be rooted. Each of the vertices of the coarse grid octants, which overlap with the fine grid octants surrounding this fine grid vertex, can be mapped to one of these 1758 possibilities. It is also possible that some of these vertices are mapped to the same location. When we pre-compute the masks described earlier, we want to identify these many-to-one mappings and only one of them is selected to contribute to the fine grid node under consideration. Details on identifying these cases are given in [19].

[7] This is a weak upper bound.

We can not pre-compute the masks offline since this depends on the coarse and fine octrees under consideration. To do this computation efficiently, we employ a 'dummy' MatVec before we actually begin solving the problem. In this dummy MatVec, we use a set of 16 bytes per fine grid node; 2 bytes for each of the eight fine grid octants surrounding the node. In these 16 bits, we store the flags to determine the following: **(1)** Whether or not the coarse and fine grid octants are the same (1 bit). **(2)** The child number of the current fine grid octant (3 bits).**(3)** The child number of the corresponding coarse grid octant (3 bits). **(4)** The hanging configuration of the corresponding coarse grid octant (5 bits). **(5)** The relative size of the current fine grid octant with respect to the reference element (2 bits).

Using this information and some simple bitwise operations, we can compute and store the masks for each fine grid node. The dummy MatVec is an operation with $\mathcal{O}(N)$ work complexity and has an $\mathcal{O}(\frac{N}{n_p})$ parallel time complexity.

Integrid transfers. Unlike the finite element MatVec the loop is split into three parts in this case. We can not loop over ghost octants in this case. This is because the ghost octants on the coarse and fine grids need not be aligned. Hence, each CPU only loops 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. We first loop over some of the elements in the interior of the CPU domains since these elements do not share any nodes with the neighboring CPUs. Simultaneously, each CPUs also reads the ghost values from the other CPUs in the background. At the end of the first loop, we use the ghost values that were received from the other CPUs and loop over those elements that will contribute to ghost values. At the end of the second loop, we exchange the updated ghost values and simultaneously loop over the remaining elements in the interior of the CPU domains.

2.4 Matrix vector multiplication

Every octant is owned by a single CPU. However, the values of unknowns associated with octants on inter-CPU boundaries need to be shared among several CPUs. We keep multiple copies of the information related to these octants and we term them 'ghost' octants. In our implementation of the finite element MatVec, each CPU iterates over all the octants it owns and also loops over a layer of ghost octants that contribute to the nodes it owns. Within the loop, 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 pre-computed stencils. We then use the selected stencil in a standard element based assembly technique. Although the CPUs need to read ghost values from other CPUs they only need to write data back to the nodes they own and do not need to write to ghost nodes. Thus, there is only one communication step within each MatVec. We even overlap this communication with useful computation. We first loop over the elements in the interior of the CPU domain since these elements do not share any nodes with the neighboring CPUs. While we perform this computation, we communicate the ghost values in the background. At the end of the first loop, we use the ghost values that were received from the other CPUs and loop over the remaining elements.

Algorithm 2 Adaptive Mesh Refinement

1. Coarsen or refine an octree using the exact analytical solution at the current time step
2. Balance the octree to enforce the 2:1 balance constraint
3. Mesh the octree to get the element-node connectivity
4. Transfer the solution at the previous time step to the new mesh
5. Solve the linear system of equations using CG with Jacobi preconditioner

Algorithm 3 Solution transfer algorithm

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

2.5 Adaptive mesh refinement

We solve a linear parabolic problem: $\frac{\partial u}{\partial t} = \Delta u + f(\mathbf{x}, t)$ with homogeneous Neumann boundary conditions using adaptive mesh refinement. We compute $f(\mathbf{x}, \mathbf{t})$ using an analytical solution of a traveling wave of the form $u(\mathbf{x}, t) = \exp(-10^4(y - 0.5 - 0.1t - 0.1\sin(2\pi x))^2)$ (see Figure 7). A second order implicit Crank-Nicholson scheme is used to solve the problem for 10 time steps with a step size of $\delta t = 0.05$. We used CG with Jacobi preconditioner to solve the linear system of discretized equations at every time step. We build the octree using the exact analytical solution. We do not use any error estimator. Our focus is in demonstrate the performance of our tree-construction, balancing, meshing and solution transfer schemes. In Algorithm 2, we describe the adaptive mesh refinement scheme. We also present a parallel algorithm to map the solution between the meshes in Algorithm 3. The overall computational complexity of the solution transfer algorithm, assuming that the octrees have $\mathcal{O}(N)$ nodes and are similar, is $\mathcal{O}(N/n_p + n_p \log n_p)$ We do not require the meshes at two different time steps to be aligned or to share the same partition. In fact, the two meshes can be entirely different. Of course, the greater their differences the higher the intergrid transfer communication costs.

2.6 Summary of the overall multigrid algorithm

1. A 'sufficiently' fine[8] 2:1 balanced complete linear octree is constructed using the algorithms described in [22].

2. Starting with the finest octree, a sequence of 2:1 balanced coarse linear octrees is constructed using the global coarsening algorithm.

3. Starting with the coarsest octree, the octree at each level is meshed using the algorithm described in [21]. As long as the load imbalance across the CPUs is acceptable and as long as the coarser grid was able to be partitioned without leaving any CPU idle, the partition of the coarser grid is imposed on to the finer grid during meshing. If either of

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

the above two conditions is violated then the octree for the finer grid is duplicated; One of them is meshed using the partition of the coarser grid and the other is meshed using a fresh partition computed using the block partition algorithm. The process is repeated until the finest octree has been meshed.

4. A dummy restriction MatVec (Section 2.3.1) is performed at each level (except the coarsest) and the masks that will be used in the actual restriction and prolongation MatVecs are computed and stored.

5. For the case of variable coefficient operators, vectors that store the material properties at each level are created.

6. The discrete system of equations is then solved using the conjugate gradient algorithm preconditioned with the multigrid scheme.

Complexity. Let N be the total number of octants in the fine level and n_p be the number of CPUs. In [21] and [22], we showed that the parallel complexity of the single-level construction, 2:1 balancing, partition, and meshing is $\mathcal{O}(\frac{N}{n_p}\log\frac{N}{n_p} + n_p \log n_p)$ for trees that are nearly uniform. The cost of a MatVec is $\mathcal{O}(\frac{N}{n_p})$.

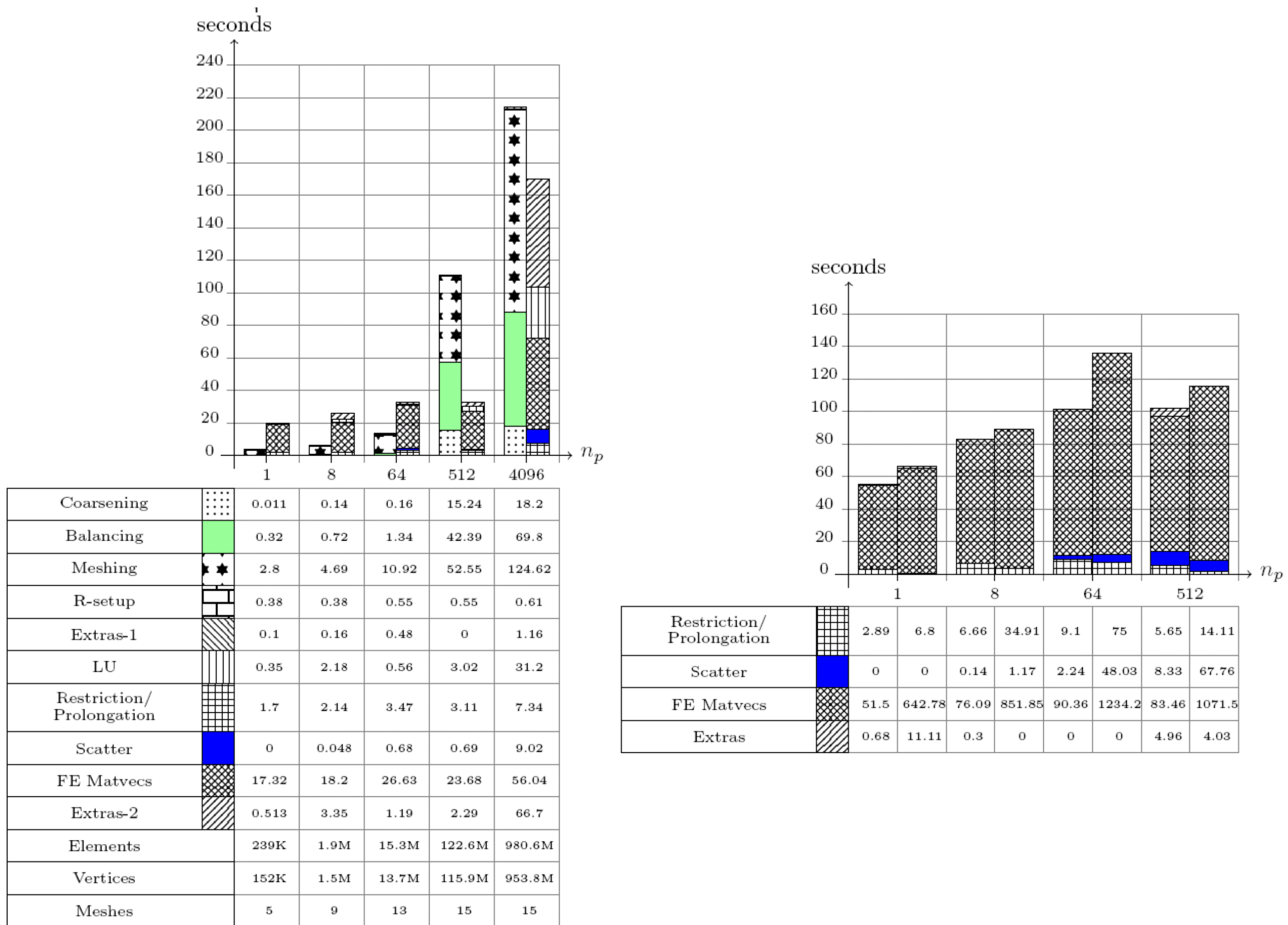
For the multigrid case, we report the complexity (again for trees that are nearly uniform) for the case in which all CPUs are used in all levels. Given that complexity of the a single-level coarsening is $\mathcal{O}(\frac{N}{n_p})$, the overall complexity of the coarsening is equal to $\sum_{k=0:L}\frac{N_k}{n_p}$. For a regular grid and the coarse grid had size 1, $L = \log N/3$ and $N_k = 2^{-3k}N$. Thus the overall coarsening complexity is the same of a single-level coarsening. Using a similar argument the cost for balancing and meshing is $\mathcal{O}(\frac{N}{n_p}\log(\frac{N}{n_p}) + n_p\log n_p)L$. (Notice the L factor in the communication cost.)

This estimate does not include the communication costs related to the mapping between different partitionings of the same octree. This operation is carried using and `MPI_Alltoallv()` call, which has and $\mathcal{O}(mp)$ complexity, with m being the message length [12]. We have not been able to derive the cost this operation. Notice that in most cases the repartition octree has significant overlaps since it is always Morton-ordered and the Block Partitioning algorithm uses this ordering. Also, out empirical results show that the associated costs are not two high even in the case in which we repartition at each level and large number of CPUs have zero elements.

3 Numerical Experiments

In this section, we report the results from iso-granular (weak scaling) and fixed size (strong scaling) scalability experiments on up to 4K CPUs on NCSA's Intel 64 cluster and PSC's Cray XT3. The NCSA machine has 8 CPUs/node and the PSC machine has 2 CPUs/node. Isogranular scalability analysis was performed by roughly keeping the problem size per CPU fixed while increasing the number of CPUs. Fixed-size scalability was performed by keeping problem size constant and increasing the number of CPUs.

Implementation details. Here, we discuss features that we plan to incorporate into `Dendro` in the near future. Currently, the number of multigrid levels is user-specified. However, the problem size for the coarsest grid is not known



	1	8	64	512	4096
Coarsening	0.011	0.14	0.16	15.24	18.2
Balancing	0.32	0.72	1.34	42.39	69.8
Meshing	2.8	4.69	10.92	52.55	124.62
R-setup	0.38	0.38	0.55	0.55	0.61
Extras-1	0.1	0.16	0.48	0	1.16
LU	0.35	2.18	0.56	3.02	31.2
Restriction/ Prolongation	1.7	2.14	3.47	3.11	7.34
Scatter	0	0.048	0.68	0.69	9.02
FE Matvecs	17.32	18.2	26.63	23.68	56.04
Extras-2	0.513	3.35	1.19	2.29	66.7
Elements	239K	1.9M	15.3M	122.6M	980.6M
Vertices	152K	1.5M	13.7M	115.9M	953.8M
Meshes	5	9	13	15	15

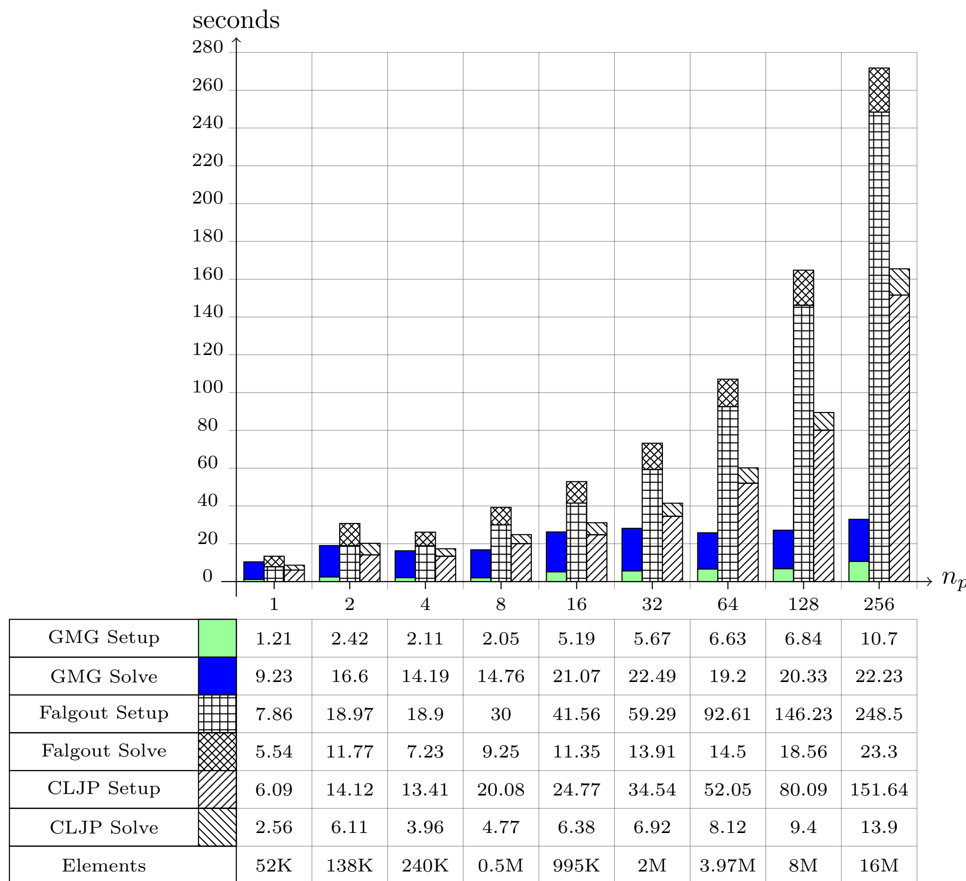
	1		8		64		512	
Restriction/ Prolongation	2.89	6.8	6.66	34.91	9.1	75	5.65	14.11
Scatter	0	0	0.14	1.17	2.24	48.03	8.33	67.76
FE Matvecs	51.5	642.78	76.09	851.85	90.36	1234.2	83.46	1071.5
Extras	0.68	11.11	0.3	0	0	0	4.96	4.03

Figure 3: *The LEFT FIGURE shows isogranular scalability with a grain size of* $0.25M$ *(approx) elements per CPU* (n_p) *on the finest level. The difference between the minimum and maximum levels of the octants on the finest grid is 5. A V-cycle using 4 pre-smoothing steps and 4 post-smoothing steps per level was used as a preconditioner to CG. The damped Jacobi method with a damping factor of 0.857 was used as the smoother at all multigrid levels. A relative tolerance of* 10^{-10} *in the 2-norm of the residual was used. 6 CG iterations were required in each case, to solve the problem to the specified tolerance. The finest level octrees for the multiple CPU cases were generated using regular refinements from the finest octree for the single CPU case. SuperLU was used to solve the coarsest grid problem. This isogranular scalability experiment was performed on NCSA's Intel 64 cluster.*
The RIGHT FIGURE gives the results for the variable coefficient scalar Laplacian operator (left column) and results for the constant coefficient linear elastostatic problem (right column). For the elasticity problem, the actual timings are given in the table below and are $1/10$ *of that value is plotted. The coefficient of the scalar Laplacian operator was chosen to be* $(1 + 10^6(\cos^2(2\pi x) + \cos^2(2\pi y) + \cos^2(2\pi z)))$. *The elastostatic problem uses homogeneous Dirichlet boundary conditions and the scalar problem uses homogeneous Neumann boundary conditions. A Poisson's ratio of 0.4 was used for this problem. The solver options are the same as described in the left figure. Only the timings for the solve phase are reported. The timings for the setup phase are not reported since the results are similar to that the left figure. This isogranular scalability experiment was performed on PSC's Cray XT3.*

a-priori. Hence, choosing an appropriate number of multigrid levels to better utilize all the available CPUs is not obvious. We are currently working on developing heuristics to dynamically adjust the number of multigrid levels. Imposing a partition computed on the coarse grid onto a fine grid may lead to load-imbalance. However, not doing so results in duplicating meshes for the intermediate levels and also increases communication during intergrid transfer operations. Hence, a balance between the two need to be arrived at and we are currently working on developing heuristics to address this issue. In Figure 2, we depict the use of duplicate octrees that are used to map scalar and vector functions between different octree partitions. These partitions can be defined by three different ways: **(1)** One way is using the same MPI communicator

and a uniform load across all CPUs; **(2)** A second way is again using the same MPI communicator but partitioning the load across a fewer number of CPUs and letting the other CPUs idle; and **(3)** Creating a new communicator. In this set of experiments, we use the the first way, i.e.,**we partition across the total number of CPUs for all levels. Also, we repartition at each level.** In this way, the results reported in the isogranular and fixed-size scalability results represent the worst-case performance for our method.

Isogranular scalability. In Figure 3, we report results for the constant-coefficient (left subfigure) and variable-coefficient (right subfigure) scalar elliptic problem with homogeneous Neumann boundary conditions on meshes whose mesh-size follows a Gaussian distribution. The coarsest level uses 1 CPU only in all of the experiments unless otherwise specified. We use `SuperLU_dist` [17] as an exact solver at the coarsest grid. 4 multigrid levels were used on 1 CPU and the number of multigrid levels were incremented by 1 every time the number of CPUs increased by a factor of 8. In the left subfigure, for each CPU, the left column gives the time for the setup-phase and the right column gives the time for the solve-phase.



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

Figure 4: *A variable coefficient (contrast of 10^7) elliptic problem with homogeneous Neumann boundary conditions was solved on meshes constructed on Gaussian distributions. A 7-level multiplicative-multigrid cycle was used as a preconditioner to CG for the GMG scheme. SuperLU_Dist was used to solve the coarsest grid problem in the GMG scheme.*

The setup cost for the GMG scheme includes the time for constructing the mesh for all the levels (including the finest), constructing and balancing all the coarser levels, setting up the intergrid transfer operators by performing one dummy

Restriction MatVec at each level. Here we repartition at each level. The time to create the work vectors for the MG scheme and the time to build the coarsest grid matrix are included in 'Extras-1'. 'Scatter' refers to the process of transferring the vectors between two different partitions of the same multigrid level during the intergrid transfer operations. This is required whenever the coarse and fine grids do not share the same partition. The time spent in applying the Jacobi preconditioner, computing the inner-products within CG and solving the coarsest grid problems using LU are all grouped together as 'Extras-2'. In Figure 4, we report results from a comparison with the algebraic multigrid scheme. In order to minimize communication costs, the coarsest level was distributed on a maximum of 32 CPUs in all experiments.

For BoomerAMG, we experimented with two different coarsening schemes: Falgout coarsening and CLJP coarsening. The results from both experiments are reported. [14] reports that Falgout coarsening works best for structured grids and CLJP coarsening is better suited for unstructured grids. Since octree meshes lie in between both structured and generic unstructured grids, we compare our results using both the schemes. Both GMG and AMG schemes used 4 pre-smoothing steps and 4 post-smoothing steps per level with the damped Jacobi smoother. A relative tolerance of 10^{-10} in the 2-norm of the residual was used in all the experiments.

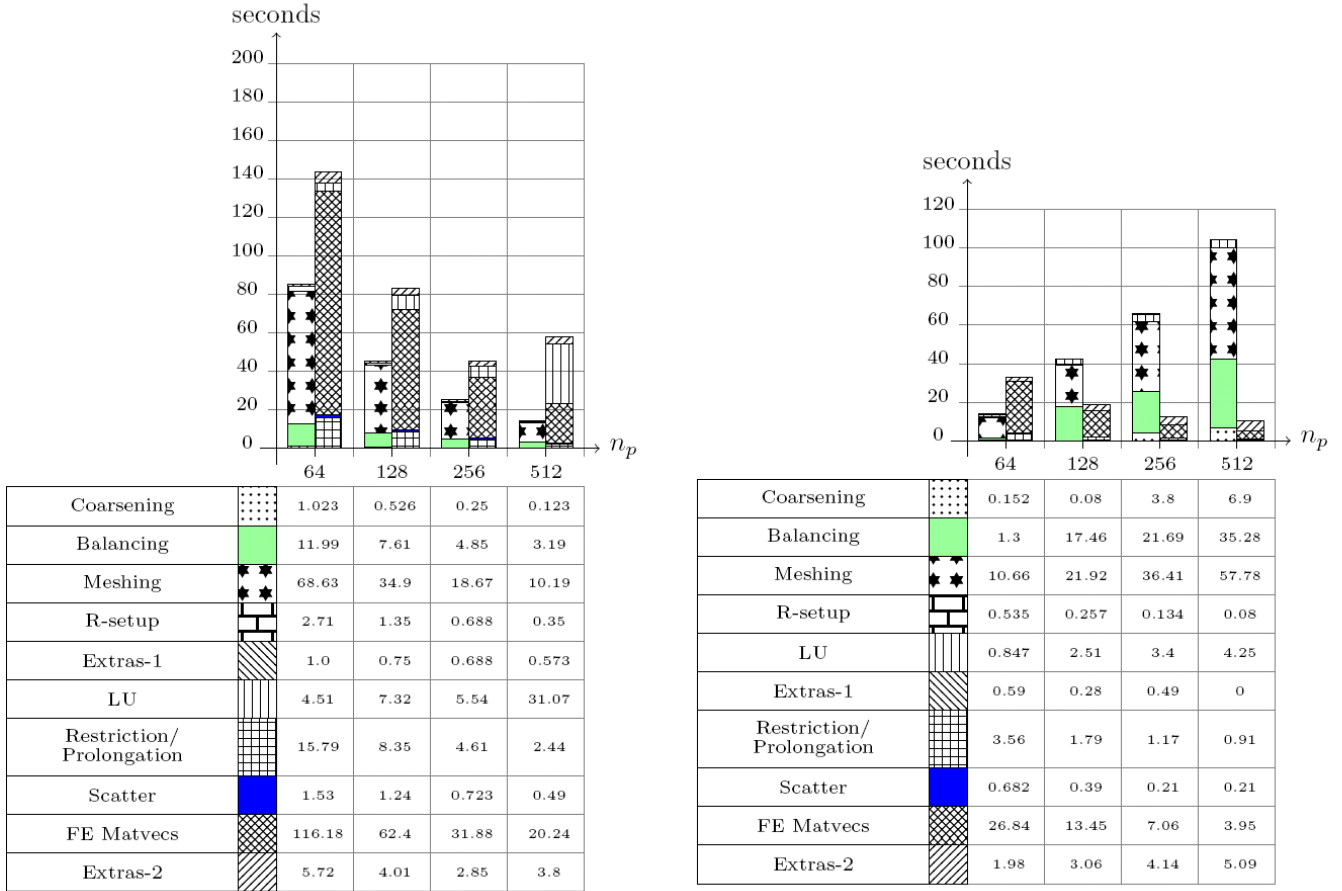
The GMG scheme took about 12 CG iterations, the Falgout scheme took about 7 iterations and the CLJP scheme also took about 7 iterations. Each node of the cluster has 8 CPUs which share an 8GB RAM. However, only 1 CPU per node was utilized in the above experiments. This is because the AMG scheme required a lot of memory and this allowed the entire memory on any node to be available for a single process.[9] The setup time reported for the AMG schemes includes the time for meshing the finest grid and constructing the finest grid FE matrix, both of which are quite small (≈ 1.35 seconds for meshing and ≈ 22.93 seconds for building the fine grid matrix even on 256 CPUs) compared to the time to setup the rest of the AMG scheme. Although GMG seems to be performing well, more difficult problems with multiple discontinuous coefficients can cause it to fail. *Our method is not robust in the presence of discontinuous coefficients—in contrast to AMG.*

Fixed-size scalability. Next, we report fixed-size scalability results for the Poisson and elasticity solvers. [10] In Figure 5, we report fixed-size scalability for two different grain sizes for the Poisson problem. In Figure 6 we report fixed-size scalability results of the elasticity and variable-coefficient Poisson cases. Overall we observe excessive costs for the coarsening, balancing, and meshing when the grain size is relatively small. Like in the isogranular case, we repartition at each multigrid level and we use all available CPUs. However, notice that the MatVecs and overall the solver scale quite well.

AMR. Finally, in Table 2 and Figure 7, we report the performance of the balancing, meshing and the solution transfer algorithms to solve the linear parabolic problem described in Section 2.5.

[9] We did not attempt to optimize neither our code nor AMG and we used the default options. It is possible that one can reduce the AMG cost by using appropriate options.

[10] The elastostatics equation is given by $\mathrm{div}(\mu(x)\nabla u(x)) + \nabla(\lambda(x) + \mu(x))\nabla v) = b(x)$, where λ is a scalar field; and v and b are 3D vector functions.



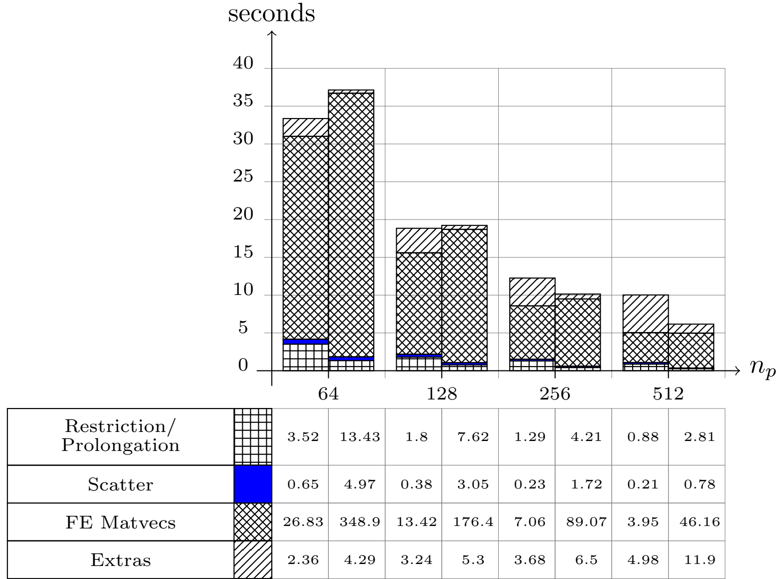
	64	128	256	512
Coarsening	1.023	0.526	0.25	0.123
Balancing	11.99	7.61	4.85	3.19
Meshing	68.63	34.9	18.67	10.19
R-setup	2.71	1.35	0.688	0.35
Extras-1	1.0	0.75	0.688	0.573
LU	4.51	7.32	5.54	31.07
Restriction/ Prolongation	15.79	8.35	4.61	2.44
Scatter	1.53	1.24	0.723	0.49
FE Matvecs	116.18	62.4	31.88	20.24
Extras-2	5.72	4.01	2.85	3.8

	64	128	256	512
Coarsening	0.152	0.08	3.8	6.9
Balancing	1.3	17.46	21.69	35.28
Meshing	10.66	21.92	36.41	57.78
R-setup	0.535	0.257	0.134	0.08
LU	0.847	2.51	3.4	4.25
Extras-1	0.59	0.28	0.49	0
Restriction/ Prolongation	3.56	1.79	1.17	0.91
Scatter	0.682	0.39	0.21	0.21
FE Matvecs	26.84	13.45	7.06	3.95
Extras-2	1.98	3.06	4.14	5.09

Figure 5: *LEFT FIGURE: Scalability for a fixed fine grid problem size of 64.4M elements. The problem is the same as described in Figure 3. 8 multigrid levels were used. 64 CPUs were used for the coarsest grid for all cases in this experiment. The minimum and maximum levels of the octants on the finest grid are 3 and 18, respectively. 5 iterations were required to solve the problem to a relative tolerance of* 10^{-10} *in the 2-norm of the residual.*
RIGHT FIGURE: Scalability for a fixed fine grid problem size of 15.3M elements. The setup cost and the cost to solve the constant coefficient poisson problem are reported. 6 multigrid levels were used. This fixed size scalability experiment was performed on NCSA's Intel 64 cluster. This fixed size scalability experiment was performed on NCSA's Intel 64 cluster.

Table 2: *Isogranular scalability with a grain size of 38K (approx) elements per CPU (n_p). A constant-coefficient linear parabolic problem with homogeneous Neumann boundary conditions was solved on octree meshes. We used the analytical solution of a traveling wave of the form* $\exp(-10^4(y - 0.5 - 0.1t - 0.1\sin(2\pi x))^2)$ *to construct the octrees. We used a time step of* $\delta t = 0.05$ *and solved for 10 time steps. We used second-order Crank-Nicholson scheme for time-stepping and CG with Jacobi preconditioner to solve the linear system of equations at every time step. We used a stopping criterion of* $\|r\|/\|r_0\| < 10^{-6}$*. We report the number of elements (***Elements***), CG iterations (***iter***), Solve time (***Solve***), total time to balance (***Balancing***) and mesh (***Meshing***) the octrees generated at each time step. We also report the time to transfer the solution between the meshes (***Transfer***) at two different time steps. The total number of elements and the number of CG iterations are approximately constant over all time steps. This isogranular scalability experiment was performed on NCSA's Intel 64 cluster.*

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



	64		128		256		512	
Restriction/ Prolongation	3.52	13.43	1.8	7.62	1.29	4.21	0.88	2.81
Scatter	0.65	4.97	0.38	3.05	0.23	1.72	0.21	0.78
FE Matvecs	26.83	348.9	13.42	176.4	7.06	89.07	3.95	46.16
Extras	2.36	4.29	3.24	5.3	3.68	6.5	4.98	11.9

Figure 6: *Scalability for a fixed fine grid problem with 15.3M elements. The cost to solve the variable coefficient poisson problem and the constant coefficient linear elasticity problem are reported. The left column on each CPU gives the results for the variable coefficient scalar Laplacian operator and the right column gives the results for the constant coefficient linear elastostatic problem. For the elasticity problem, the actual timings are given in the table but are scaled by an order of magnitude (1/10) for the plot. The setup cost is the same as in Fig. 5. 6 multigrid levels were used. This fixed size scalability experiment was performed on NCSA's Intel 64 cluster.*

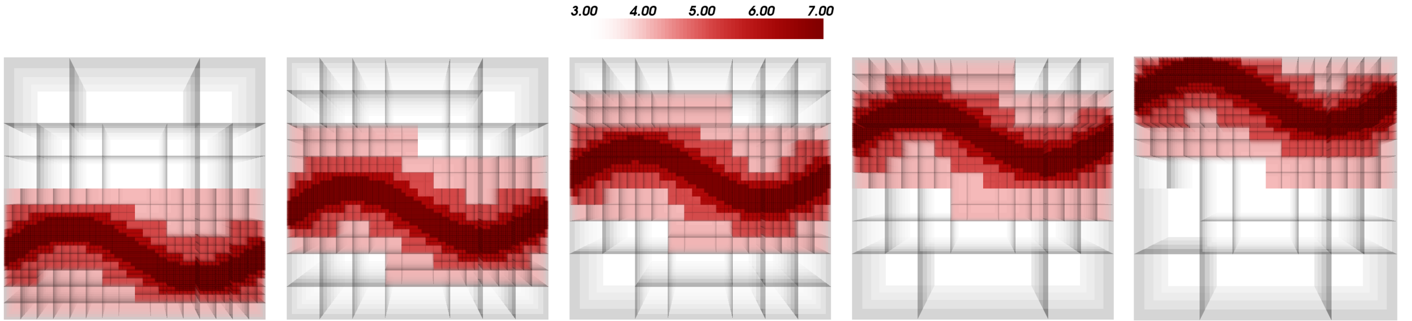


Figure 7: *An example of adaptive mesh refinement on a traveling wave. This is a synthetic solution which we use to illustrate the adaptive octrees. The coarsening and refinement are based on the approximation error between the discretized and the exact function.*

4 Conclusions

We have described a parallel geometric multigrid method for solving partial differential equations using finite elements on octrees. Also, we described and AMR scheme that can be used to transfer vector and scalar functions between arbitrary octrees.

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

In the final submission, we will include additional results with significant improvements for meshing and balancing parts in the multigrid case, to have the load determined by a minimum grain size.

We have demonstrated reasonable scalability of our implementation and can solve problems with billions of elements on thousands of CPUs. We tested the worse case scenarios for our code in which we repartition at each level and we use all available CPUs for all levels. We have demonstrated that our implementation works well even on problems with largely varying material properties. We have compared our geometric multigrid implementation with a state-of-the-art algebraic multigrid implementation in a standard off-the-shelf package (HYPRE). Overall we showed that the proposed algorithm is quite efficient although significant work remains to improve the partitioning algorithm and the overall robustness of the scheme in the presence of discontinuous coefficients.

References

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

[2] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, *PETSc home page*, 2001. http://www.mcs.anl.gov/petsc.

[3] G. T. Balls, S. B. Baden, and P. Colella, *SCALLOP: A highly scalable parallel Poisson solver in three dimensions*, in SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, Washington, DC, USA, 2003, IEEE Computer Society, p. 23.

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

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

[6] M. W. Bern, D. Eppstein, and S.-H. Teng, *Parallel construction of quadtrees and quality triangulations*, International Journal of Computational Geometry and Applications, 9 (1999), pp. 517–532.

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

[8] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco, *Dynamic octree load balancing using space-filling curves*, Tech. Report CS-03-01, Williams College Department of Computer Science, 2003.

[9] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. M. Yang, *A survey of parallelization techniques for multigrid solvers*, in Parallel Processing for Scientific Computing, M. A. Heroux, P. Raghavan, and H. D. Simon, eds., Cambridge University Press, 2006, pp. 179–195.

[10] R. Falgout, *An introduction to algebraic multigrid*, Computing in Science and Engineering, Special issue on Multigrid Computing, 8 (2006), pp. 24–33.

[11] R. Falgout, J. Jones, and U. Yang, *The design and implementation of hypre, a library of parallel high performance preconditioners*, in Numerical Solution of Partial Differential Equations on Parallel Computers, A. Bruaset and A. Tveito, eds., vol. 51, Springer-Verlag, 2006, pp. 267–294.

[12] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *An Introduction to Parallel Computing: Design and Analysis of Algorithms*, Addison Wesley, second ed., 2003.

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

[14] V. E. Henson and U. M. Yang, *Boomeramg: a parallel algebraic multigrid solver and preconditioner*, Appl. Numer. Math., 41 (2002), pp. 155–177.

[15] F. H ulsemann, M. Kowarschik, M. Mohr, and U. R ude, *Parallel geometric multigrid*, in Numerical Solution of Partial Differential Equations on Parallel Computers, A. M. Bruaset and A. Tveito, eds., Birk auser, 2006, pp. 165–208.

[16] A. Jones and P. Jimack, *An adaptive multigrid tool for elliptic and parabolic systems*, International Journal for Numerical Methods in Fluids, 47 (2005), pp. 1123–1128.

[17] X. S. Li and J. W. Demmel, *SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems*, ACM Transactions of Mathematical Software, 29 (2003), pp. 110–140.

[18] D. J. Mavriplis, M. J. Aftosmis, and M. Berger, *High resolution aerospace applications using the NASA Columbia Supercomputer*, in SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, Washington, DC, USA, 2005, IEEE Computer Society, p. 61.

[19] R. Sampath and G. Biros, *A parallel geometric multigrid method for finite elements on octree meshes*, tech. report, University of Pennsylvania, 2008. Sumbitted for publication.

[20] R. Sampath, H. Sundar, S. S. Adavani, I. Lashuk, and G. Biros, `Dendro` *home page*, 2008. http://www.seas.upenn.edu/∼csela/dendro.

[21] H. Sundar, R. Sampath, C. Davatzikos, and G. Biros, *Low-constant parallel algorithms for finite element simulations using linear octrees*, in Proceedings of SC2007, The SCxy Conference series in high performance networking and computing, Reno, Nevada, 2007, ACM/IEEE.

[22] H. Sundar, R. S. Sampath, and G. Biros, *Bottom-up construction and 2:1 balance refinement of linear octrees in parallel*, SIAM Journal on Scientific Computing, (2008). to appear.

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

[24] T. Tu, D. R. O'Hallaron, and O. Ghattas, *Scalable parallel octree meshing for terascale applications*, in SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, Washington, DC, USA, 2005, IEEE Computer Society, p. 4.

[25] 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 SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, New York, NY, USA, 2006, ACM Press, p. 91.