Low-constant Parallel Algorithms for Finite Element Simulations using Linear Octrees

Hari Sundar
Department of Bioengineering
University of Pennsylvania
Philadelphia, PA
hsundar@seas.upenn.edu

Rahul S. Sampath
Department of MEAM[*]
University of Pennsylvania
Philadelphia, PA
rahulss@seas.upenn.edu

Santi S. Adavani
Department of MEAM
University of Pennsylvania
Philadelphia, PA
adavani@seas.upenn.edu

Christos Davatzikos
Department of Radiology
University of Pennsylvania
Philadelphia, PA
christos@rad.upenn.edu

George Biros
Department of MEAM
University of Pennsylvania
Philadelphia, PA
biros@seas.upenn.edu

ABSTRACT

In this article we propose parallel algorithms for the construction of conforming finite-element discretization on linear octrees. Existing octree-based discretizations scale to billions of elements, but the complexity constants can be high. In our approach we use several techniques to minimize overhead: a novel bottom-up tree-construction and 2:1 balance constraint enforcement; a Golomb-Rice encoding for compression by representing the octree and element connectivity as an Uniquely Decodable Code (UDC); overlapping communication and computation; and byte alignment for cache efficiency. The cost of applying the Laplacian is comparable to that of applying it using a direct indexing regular grid discretization with the same number of elements. Our algorithm has scaled up to four billion octants on 4096 processors on a Cray XT3 at the Pittsburgh Supercomputing Center. The overall tree construction time is under a minute in contrast to previous implementations that required several minutes; the evaluation of the discretization of a variable-coefficient Laplacian takes only a few seconds.

1. INTRODUCTION

In this article we propose parallel linear octree algorithms for tree construction, two-one balancing, and discretization of partial differential equations using conforming trilinear finite elements. Typical approaches for large-scale discretizations include logically structured grids, block structured and overlapping grids, unstructured grids, and octrees. All methods have advantages and disadvantages. For example, structured grids are relatively easy to implement, have low mem-

---

[*]Mechanical Engineering and Applied Mechanics

ory requirements, and avoid indirect memory references. Structured grids however, limit adaptivity; for certain problems this limitation can result in excessively large systems of equations. Although unstructured meshes can conform to complex geometries and enable non-uniform discretizations, they incur the overhead of having to explicitly store element-node connectivity information and in general being cache inefficient because of random access [3, 12, 25]. Octrees offer a good balance between adaptivity and efficient performance. Optimal complexity scalable implementations have been developed [22], but certain parts of the existing algorithms have high constants. In this article we propose novel algorithms for octree based meshing that achieve lower wall-clock running times.

Related work. There is a large literature on large-scale finite element (FEM) calculations. Here we review recent work that has scaled to thousands of processors. One of the largest calculations was reported in [6]. The authors proposed a scheme for conforming discretizations and multigrid solvers on semi-structured meshes. Their approach is highly scalable for nearly structured meshes and for constant coefficient PDEs. However, it does not support adaptive discretizations, and its computational efficiency diminishes in the case of variable coefficient operators. Examples of scalable approaches for unstructured meshes include [1] and [16]. 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: setting up the mesh and building the associated linear operators can take thousands of seconds. A significant part of CPU time is related to the multigrid scheme (we do not consider multigrid in this paper); even in the single grid cases however, the run times are quite large.

The high-costs related to partitioning, setup, and accessing generic unstructured grids, has motivated the design of octree-based data structures. Such constructions have been used in sequential and modestly parallel adaptive finite element implementations [5, 11, 14, 17], and many-body algorithms [10, 13, 26, 27]. State-of-the-art in-core and out-of-

core implementations for finite elements can be found in [2, 20, 21, 22, 23]. In these papers the authors propose a top-down octree construction resulting in a Morton-ordered tree, followed by a 2:1 balance constraint using a novel algorithm (the parallel prioritized ripple propagation). Once the tree is constructed, a finite-element framework is built on top of the tree structure. Evaluating the discrete Laplacian operators (a matrix-free matrix-vector multiplication) requires three tree traversals, one traversal over the elements, and two traversals to apply projection over the so-called "hanging nodes", to enforce a conforming discretization [14].

Contributions. Our goal is to significantly improve the construction, balancing, and discretization performance of octree-based algorithms. In our recent work [19], we present novel algorithms for bottom-up construction and 2:1 balance refinement of large linear octrees on distributed memory machines. In the present work, we build upon these algorithms and develop efficient data structures that support trilinear conforming finite element calculations on linear octrees. We focus on reducing (1) the time to build these data structures; (2) the memory overhead associated in storing them; and (3) the time to perform finite element calculations using these data structures.

We avoid using multiple passes (projections) to enforce conformity; instead, we perform a single traversal by mapping each octant/element to one of eight pre-computed element types, depending on the configuration of hanging nodes for that element. Our data structure does not allow efficient random queries in the octree, but such access patterns are not necessary for FEM calculations.

The memory overhead associated with unstructured meshes arises from the need to store the element connectivity information. In regular grids such connectivity is not necessary as the indexing is explicit. For general unstructured meshes of trilinear elements one has to directly store eight indices defining the element nodes. In octrees we still need to store this information, but it turns out that instead of storing eight integers (32 `bytes`), we need to store only 12 `bytes`. We use the Golomb-Rice encoding scheme to compress the element connectivity information and to represent it as a Uniquely Decodable Code (UDC) [15]. In addition, the linear octree is stored in a compressed form that requires only one byte per octant (the level of the octant).

Finally, we employ overlapping of communication and computation to efficiently handle octants shared by several processors or *"ghost"* octants.[1] In addition, the Morton-ordering offers reasonably good memory locality. The cost of applying the discretized Laplacian operator using our approach is comparable to that of a discretization on a regular grid (without stored indexing). Thus, our approach offers significant savings (over structured grids and general unstructured grids) in the case of non-uniform discretizations.

---

[1] Every octant is owned by a single processor. However, the values of unknowns associated with octants on interpocessor 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 a nutshell, our contributions are the following: we present (1) a bottom-up algorithm to construct a linear octree from a given set of randomly partitioned points or leaf-octants; (2) a 2:1 balancing algorithm; (3) an algorithm to build element-to-nodal connectivity information efficiently; (4) a compression scheme for the octree and the element connectivity that achieves a three-fold compression (a total of four words per octant); and (5) a lookup table based conforming discretization scheme that requires only a single traversal for the evaluation of a partial differential operator. All five stages are parallel. Our approach results in symmetric, second-order accurate discretizations of self-adjoint operators. Our algorithms have $\mathcal{O}(n \log n)$ work and $\mathcal{O}(n)$ storage complexity. For typical distributions of octants (and work per octant), the parallel time complexity of our scheme is $\mathcal{O}(n/n_p \log(n/n_p) + n_p \log n_p)$, where n is the final number of leaves and n_p is the number of processors. In contrast to existing implementations, our methods avoid iterative communications and thus, achieve low absolute runtime and excellent scalability.

Our algorithm has scaled to four billion octants on 4096 processors on a Cray XT3 ("Big Ben") at the Pittsburgh Supercomputing Center. The overall time for octree construction, balancing, and meshing is slightly over a minute; a second-order accurate evaluation of the discrete Laplacian takes only a few seconds. Our experiments demonstrate that the algorithms proposed in this paper achieve a significantly lower running time compared to previous implementations.

Organization of the paper. The rest of this paper is organized as follows: in Section 2 we summarize definitions related to octrees; in Section 3 we summarize our previous work on octree construction and balancing, which lays the foundation for the present work. In Section 4 we describe octree meshing, and the octree and mesh compression. In Section 5 we describe how we perform the finite element computation. In Section 6 we present performance results that demonstrate the efficiency of our implementation.

2. BACKGROUND

Octrees are trees in which every node has a maximum of eight children. They are analogous to binary trees (maximum of 2 children per node) in 1-D and quadtrees (maximum of 4 children per node) in 2-D. A node of an octree is called an octant. An octant with no children is called a *leaf*. The only octant with no parent is the *root*. Octants that have the same parent are called *siblings*. An octant's children, grandchildren and so on and so forth are collectively referred to as the octant's *descendants* and this octant will be an *ancestor* of its descendants. The depth of an octant from the root is referred to as its *level*. As shown in Figure 1(a), the root of the tree is at level 0 and the children are one level higher than the parent.

Octrees and quadtrees can be used to partition cuboidal and rectangular regions, respectively (Figure 1(b)). These regions are referred to as the domain of the tree. A set of octants is said to be complete if the union of the regions spanned by them covers the entire domain. To reduce storage costs, only the complete list of leaf nodes is stored, i.e., as a linear octree. To use a linear representation, a *locational*
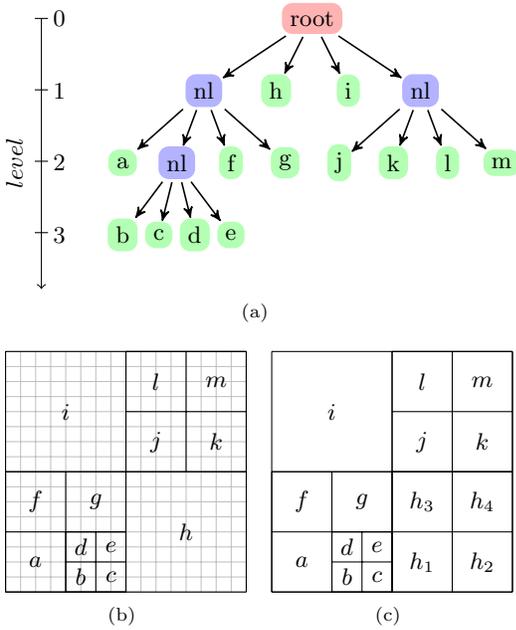
(a)

(b)

(c)

Figure 1: (a) Tree representation of a quadtree and (b) decomposition of a square domain using the quadtree, superimposed over a uniform grid, and (c) a balanced linear quadtree: result of balancing the quadtree.

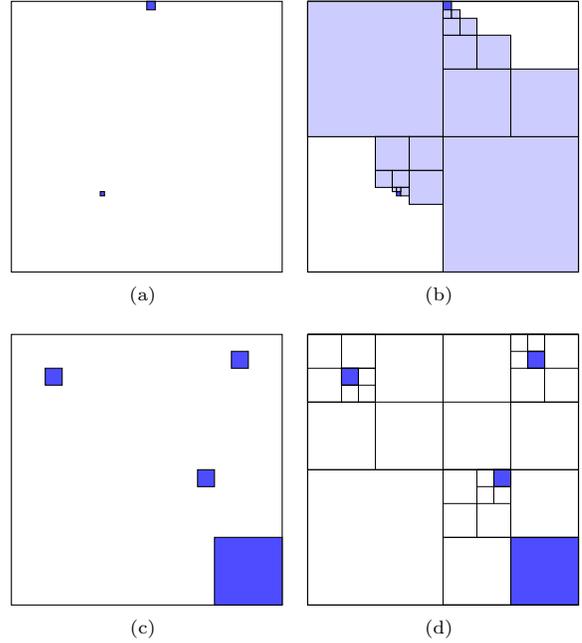

(a)   (b)

(c)   (d)

Figure 2: (b) The minimal number of octants between the cells given in (a), and (d), the coarsest possible complete linear quadtree containing all the cells in (c)

code is needed to identify the octants. A locational code is a code that contains information about the position and level of the octant in the tree. In this article we use the *Morton encoding* [8]. In the rest of the paper the terms *lesser* and *greater* are used to compare octants based on their Morton ids, and *coarser* and *finer* to compare them based on their relative sizes, i.e., their levels in the octree.

3. CONSTRUCTION AND BALANCING

In this section we descibe our algorithms for the parallel construction and balancing of linear octrees. In sections 3.1 and 3.2 we describe two algorithms, which serve as building blocks for the other algorithms described in this paper. In sections 3.3 and 3.4, we give a brief overview of our parallel construction and balancing algorithms, which are described in greater detail in [19].

3.1 Constructing complete linear octrees from a partial set of octants

In order to construct a complete linear octree from a partial set of octants (e.g. Figure 2(c)), the octants are initially sorted based on the Morton ordering and overlaps are removed.[2] Two additional octants are added to complete the domain; the first one is the coarsest ancestor of the least possible octant (the smallest descendant of the root octant at level D_{max}), which does not overlap the first given octant, and the second is the coarsest ancestor of the greatest possible octant (the largest descendant of the root octant at level D_{max}), which does not overlap the last given octant.

---

[2] We have implemented an in-house parallel sort based on a hybrid sample/biotonic sort.

The octants are distributed across the processors to get a uniform load distribution. The local complete linear octree is subsequently generated by completing the region between every consecutive pair of octants. The region between two octants, a and $b > a$, is completed by first calculating the nearest common ancestor of the octants a and b. This octant is split into its eight children. Out of these, only the octants that are either greater than a and lesser than b or ancestors of a are retained and the rest are discarded. The ancestors of either a or b are split again and we iterate until no further splits are necessary. Each processor is also responsible for completing the region between the first octant owned by that processor and the last octant owned by the previous processor, thus ensuring that a global complete linear octree is produced. Figures 2(b) and 2(d) illustrate the result of completing the morton-space between 2 given octants and the result of using this idea to produce the coarsest possible complete linear octree containing all of a given list of octants.

3.2 Partitioning Linear Octrees

Two desirable qualities of any partitioning strategy are load balancing, and minimization of overlap between the processor domains. In [19] we proposed a heuristic partitioning scheme based on the intuition that a coarse grid partition is more likely to have a smaller overlap between the processor domains as compared to a partition computed on the underlying fine grid. This algorithm comprises of 3 main stages: (1) Constructing a coarse complete linear octree that is representative of the underlying data distribution. (2) Assigning weights to each octants in the coarse octree and parti-
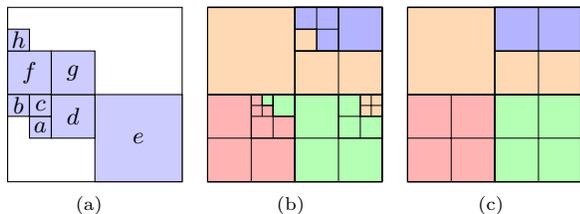
Figure 3: (a) A minimal list of quadrants covering the local domain on some processor, and (b) A Morton ordering based partition of a quadtree across 4 processors, and (c) the coarse quadrants and the final partition produced using the quadtree shown in (b).

tioning the same to achieve almost uniform load across the processors and (3) Projecting the partitioning computed in step 2 onto the original (fine) linear octree.

We sort the leaves according to their Morton ordering and then distribute them uniformly across the processors. We select the least and the greatest octant at each processor (e.g., octants a and h from Figure 3(a)) and complete the region between them, as described in Section 3.1, to obtain a list of coarse octants. We then select the coarsest cell(s) out of this list (octant e in Figure 3(a)). We use the selected octants at each processor and construct a complete linear octree as described in Section 3.1. This gives us a global coarse complete linear octree that is based on the underlying data distribution. We compute the load of each of these coarse blocks by computing the number of original octants that lie within it. The blocks are then distributed across the processors such that the total weight on each processor is roughly the same. Note that the domain occupied by the blocks and the original octants on any given processor is not the same, but it does overlap to a large extent. The overlap is guaranteed by the fact that both are sorted according to the Morton ordering and that the partitioning was based on the same weighting function (i.e., the number of original octants). The original octants are then partitioned to align with the coarse block boundaries.

3.3   Constructing linear octrees in parallel

We use a bottom-up approach for constructing complete linear octrees in parallel from a distributed set of points such that the octree satisfies the constraint that no octant should contain more than (N_{max}^p) number of points. The crux of the algorithm is to distribute the data across the processors in such a way that there is uniform load distribution across processors and the subsequent operations to build the octree can be performed by the processors independently, i.e., requiring no additional communication. Given a set of points and a user-specified maximum level, D_{max}, we convert all points into octants at the maximum depth and then we parallel partition them using the algorithm described in Section 3.2. This produces a contiguous set of coarse blocks (with their corresponding points) on each processor. The complete linear octree is generated by iterating through the blocks and by splitting them based on number of points per block. This process is continued until no further splits are required.

3.4   2:1 balancing of linear octrees in parallel

Balance refinement is the process of refining (subdividing) octants in a complete linear octree, which fail to satisfy the balance constraint defined below:

DEFINITION 1. *A linear d-dimensional tree is k-balanced if and only if, for any $l \in [1, D_{max})$, no leaf at level l shares an m-dimensional face[3] $(m \in [k, d))$ with another leaf, at level greater than $l + 1$.*

We refer to octrees that are balanced across faces as being 2-*balanced*, those that are balanced across edges and faces as 1-*balanced*, and those that are balanced across corners, edges and faces as 0-*balanced*. An example of a 0-*balanced* quadtree is shown in Figure 1(c)[4].

The octants are refined until all their descendants, which are created in the process of subdivision, satisfy the balance constraint. These subdivisions could in turn introduce new imbalances and so the process has to be repeated iteratively. The fact that an octant can affect octants not immediately adjacent to is known as the *ripple effect*. The following property allows us to decouple the problem of balancing and "contain" the ripple effect. This allows us to work on only a subset of nodes in the octree and yet ensure that the entire octree is balanced.

DEFINITION 2. *For any octant, N, we refer to the union of the domains occupied by its potential neighbor's at the same level as N as the insulation layer around octant N.*

PROPERTY 1. *No octant outside the insulation layer around octant N can force N to split (Figure 4(a)).*

The partitioning algorithm described in Section 3.2 also forms the back-bone of our balancing algorithm. The partitioning algorithm was developed with parallel balancing of linear octrees in mind and is tailored for that purpose. The balancing algorithm consists of two stages, *intra-block* and *inter-block* balancing. Intra-block balancing is sequential, and is performed using a variant of the search-free algorithm for balancing quatrees described in [7]. However, unlike [7] which produces upto 8 times the optimal number of leaves, our algorithm produces the optimal number of leaves. In [19], we show that this search-free approach performs better than the search-based "Prioritized Ripple Propagation" algorithm (PRP) [20, 22] for intra-block balancing.

PROPERTY 2. *At the end of the intra-block balancing, the decendants of a block that do not touch any of its faces are stable. These octants do not participate in the rest of the balancing process as they will neither be split by another octant nor force any octant to split.*

---

[3] A corner is a 0-dimensional face, an edge is a 1-dimensional face and a face is a 2-dimensional face.
[4] The balance algorithm used in this work is capable of *k-balancing* a given complete linear octree; we report all results for the 0-*balance* case, which is the hardest.

The inter-block boundary is also balanced in two stages: intra-processor, inter-block boundary first and then followed by the balancing across the inter-processor boundaries. A straightforward implementation of the PRP can be used to balance across the intra-processor, inter-block boundaries. In this work we use a variant of the PRP algorithm to balance across the intra-processor, inter-block boundaries. The main difference in our implementation of the PRP is that we do not require a pointer-based representation of the octree. Besides lower storage costs, this also allows us to work on incomplete domains including domains that are not simply connected. Parallel PRP can also be used to balance the inter-processor boundaries. However, this entails the use of parallel searches. The scalability of the parallel PRP is demonstrated in [22]. In [19] we described a way to create an insulation layer around each inter-processor boundary octant and thus avoid the iterative communication associated with the parallel PRP.

PROPERTY 3. *The only octants that need to be refined after the local balancing stage are those whose insulation layer is not contained entirely within the same processor. These octants are referred to as inter-processor boundary octants*[5].

The construction of the insulation layer is done in two stages (Figure 4(b)): First, every local octant on the inter-processor boundary is communicated to processors that overlap with its insulation layer. These processors can be determined by comparing the local boundary octants against the global coarse blocks. In the second stage of communication, all the local inter-processor boundary octants that overlap with the insulation layer of a remote octant received from another processor are communicated to that processor. Octants that were communicated in the first stage are not communicated to the same processor again. After this two-stage communication, each processor balances the union of the local and remote boundary octants using a sequential ripple propagation method. Upon termination only the octants spanning the original domain spanned by the processors are retained. Although there is some redundancy in the work, it is compensated by the fact that we avoid iterative communications. In [19], we show that the number of octants communicated is lower than that needs to be communicated in parallel-search based approaches.

4. OCTREE MESHING

By octree meshing we refer to the construction of a data structure on top of the linear octree that allows FEM type calculations. In this section, we describe how we construct the support data structures in order to perform the matrix-vector multiplications (`MatVec`) efficiently. The data structure is designed to be cache efficient by using a Morton ordering based element traversal, and by reducing the memory footprint using compressed representations for both the octree and the element-node connectivity tables. The algorithm for generating the mesh given a distributed, sorted, complete, balanced linear octree is outlined in Algorithm 1.

Algorithm 1. OCTREE MESHING AND COMPRESSION

Input: *A distributed sorted complete balanced linear octree, L*
Output: *Compressed Octree Mesh and Compressed Octree.*

1. *Embed L into a larger octree, O, and add boundary octants.*
2. *Identify 'hanging' nodes.*
3. *Exchange 'Ghost' octants.*
4. *Build lookup tables for first layer of octants. (Section 4.1.1)*
5. *Perform 4-way searches for remaining octants. (Section 4.1.2)*
6. *Store the levels of the octants and discard the anchors.*
7. *Compress the mesh (Section 4.2).*

### 4.1 Computing the Element to Node Mapping

The nodes (vertices) of a given element are numbered according to the Morton ordering. An example is shown in Figure 5(a). The 0-node of an element, the node with the smallest index in the Morton ordering, is also referred to as the *"anchor"* of the element. An octant's configuration with respect to its parent is specified by specifying the node that it shares with its parent. Therefore, a 3-child octant is the child that shares its third node with its parent. Nodes that exist at the center of a face of another octant are called face-hanging nodes. Nodes that are located at the center of an edge of another octant are called edge-hanging nodes.

Since all nodes, except for boundary nodes, can be uniquely associated with an element (the element with its anchor at the same coordinate as the node) we use an interleaved representation where a common index is used for both the elements and the nodes. Because of this mapping, the input balanced octree does not have any elements corresponding to the positive boundary nodes. To account for this we embed the input octree in a larger octree with maximum depth $D_{max} + 1$, where D_{max} is the maximum depth of the input octree. All elements on the positive boundaries in the input octree add a layer of octants, with a single linear pass ($\mathcal{O}(n/p)$), and a parallel sort ($\mathcal{O}(n/p \log n/p)$).

The second step in the the computation of the element-to-node mapping is the identification of hanging nodes. All nodes which are not hanging are flagged as being *nodes*. Octants that are the 0 or 7 children of their parent (a_0, a_7) can never be hanging; by default we mark them as *nodes* (see Figure 5(a)). Octants that are $3, 5, 6$ children of their parent (a_3, a_5, a_6) can only be *face hanging*, and their status is determined by a single negative search.[6] The remaining octants ($1, 2, 4$ children) are *edge hanging* and identifying their status requires three searches.

After identifying hanging nodes, we repartition the octree using the algorithm described in Section 3.2 and all octants touching the inter-processor boundaries are communicated to the neighbouring processors. On the processors that receive such octants, these are the ghost elements, and their corresponding nodes are called ghost nodes. In our imple-

---

[5]This is a bit of a misnomer since not all of these octants actually touch an inter-processor boundary.

---

[6]By *"negative"* searches we refer to searches in the $-x$ or $-y$ or $-z$ directions. We use *"positive searches"* to refer to searches along the positive directions.
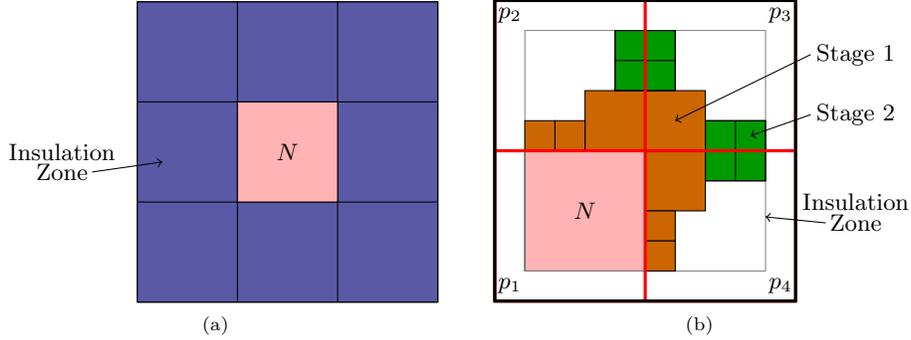
Figure 4: (a) Illustration of Property 1: No octant outside the layer of insulation can force a split on N. (b) Communication for inter-processor balancing is done in two stages: First, every octant on the inter-processor boundary (Stage 1) is communicated to processors that overlap with its insulation layer. Next, all the local inter-processor boundary octants that lie in the insulation layer of a remote octant received from another processor(N) are communicated to that processor (Stage 2).

mentation of the `MatVec`, we do not loop over ghost elements recieved from a processor with greater rank and we do not write to ghost values. This framework gives rise to a subtle special case for *singular blocks*. A singular block is a block (output of the partition algorithm), which is also a leaf in the underlying fine octree (input to the partition algorithm). If the singular block's anchor is hanging, it might point to a ghost node and if so this ghost node will be owned by a processor with lower rank. This ghost node will be the anchor of the singular block's parent. We tackle this case while partitioning by ensuring that any singular block with a hanging anchor is sent to the processor to which the first child of the singular block's parent is sent. We also send any octant that lies between (in the Morton ordering) the singular block and its parent to the same processor in order to ensure that the relative ordering of the octants is preserved.

After exchanging ghosts, we perform independent sequential searches on each processor to build the element-to-node mappings. We present two methods for the same, an exhaustive approach (Section 4.1.1) that searches for all the 8 nodes for each element, and a more efficient approach that utilizes the mapping of its negative face neighbours (Section 4.1.2) to construct its own mapping.

4.1.1 Exhaustive searches to compute mapping

The simplest approach to compute the element-to-node mapping would be to search for the nodes explicitly using a parallel search algorithm, followed by the computation of global-to-local mappings that are necessary to manage the distributed data. However, this would incur expensive communication and synchronization costs. To reduce these costs, we chose to use *a priori* communication of "ghost" octants[7] followed by independent local searches on each processor that require no communication. A few subtle cases that can not be identified easily during a priori communication are identified during the local searches and corrected for later.

For any element, all its nodes are in the positive direction (except the anchor). The exhaustive search strategy is as

---
[7]The use of blocks makes it easy to identify "ghost" octants.

follows: Generate search keys at the location of the eight nodes of the element at the maximum depth and search for them in the linear octree. Since the linear octree is sorted, the search can be performed in $\mathcal{O}(\log n)$. If the search result is a *node*, then the lookup table is updated. If we discover a hanging node instead, then a secondary search is performed to recover the correct (non-hanging) node index. As can be seen from Figure 5(a), hanging nodes are always mapped to the corresponding nodes of their parent[8]. Unfortunately, secondary searches can not be avoided despite the identification of hanging nodes prior to searching. This is because only the element whose anchor is hanging knows this information and the other elements that share this vertex must first search and find this element in order to learn this information.

Using exhaustive searches we can get the mapping for most elements, but certain special cases arise for ghost elements. This case is illustrated in Figure 5(b), where the ghost elements are drawn in red and the local elements are drawn in blue. Consider searching for the $+z$ neighbor of element a. Since a is a ghost, and we only communicate a single layer of ghost octants across processor boundaries, the node b will not be found. In such cases, we set the mapping to point to one of the hanging siblings of the missing node (c or d in this case). The most likely condition under which b in not found is when the $+z$ neighbor(s) is smaller. In this case, we know that at least one of the siblings will be hanging. Although the lookup table for this element is incorrect, we make use of the fact that the lookup table points to a non-existent node, and the owner of the true node simply copies its data value to the hanging node locations, thereby ensuring that the correct value is read. This case can only happen for ghost elements and need not be done for local elements. In addition, we occasionally observe cases where neither the searched node nor any of its siblings is found. Such cases are flagged and at the end of the lookup table construction, a parallel search is done to obtain the missing nodes directly from the processors that own them.

---
[8]The 2:1 balance constraint ensures that the nodes of the parent can never be hanging.
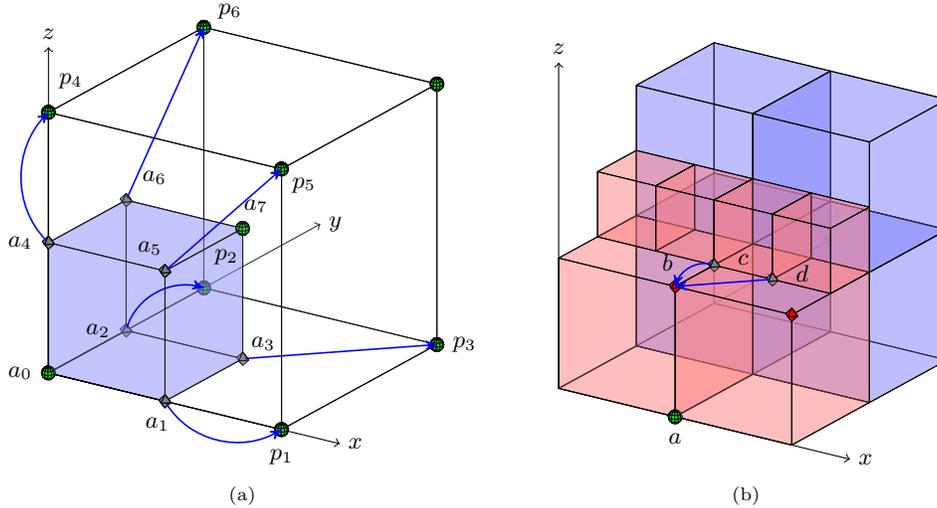
Figure 5: (a) Illustration of nodal-connectivities required to perform conforming FEM calculations using a single tree traversal. Every octant has at least 2 non-hanging nodes, one of which is shared with the parent and the other is shared amongst all the siblings. The octant shown in blue (a) is a child **0**, since it shares its zero node (a_0) with its parent. It shares node a_7 with its siblings. All other nodes, if hanging, point to the corresponding node of the parent octant instead. Nodes, a_3, a_5, a_6 are face hanging nodes and point to p_3, p_5, p_6, respectively. Similarly a_1, a_2, a_4 are edge hanging nodes and point to p_1, p_2, p_4. (b) The figure explains the special case that occurs during exhaustive searches of ghost elements. Element anchored at a, when searching for node b, will not find any node. Instead, one of the hanging siblings of b, (c, d) which are hanging will be pointed to. Since hanging nodes do not carry any nodal information, the nodal information for b will be replicated to all its hanging siblings during update for the ghosts.

4.1.2 Four-way searches to compute mapping

The exhaustive search explicitly searches for all nodes and in many cases is the only way to get the correct element to node mapping. However, it requires a minimum of 7 and a maximum of 13 searches per element. All searches are performed on a sorted list, and can be done in $\mathcal{O}(\log n)$. In order to reduce the constants associated with the exhaustive search, we use the exhaustive search only for the first layer of octants (octants that do not have neighbours in the negative x, y and z directions). For all other octants, the lookup information can be copied from the elements in the negative directions. Each element in the negative x, y and z directions that shares a face with the current element, also shares 4 nodes. Therefore, by performing negative searches along these directions, we can obtain the lookup information for 7 out of the 8 nodes of an element. Only the last node, labeled a_7 in Figure 5(a), cannot be obtained using a negative search and a positive search is required.

In order to get the mapping information using negative searches, we perform the search in the negative direction and check if the current element is a sibling of the element obtained via the negative search. If the element found by the search is not a sibling of the current element, then the lookup information can be copied via a mapping. For the example shown in Figure 6(a), given the element b and searching in the $-y$ direction, we find a, then the node mapping is $(b_0, b_1, b_4, b_5) = (a_2, a_3, a_6, a_7)$. Corresponding mappings are $(b_0, b_2, b_4, b_6) = (a_1, a_3, a_5, a_7)$, and $(b_0, b_1, b_2, b_3) = (a_4, a_5, a_6, a_7)$, for negative searches along the x and z axes, respectively. Unfortunately, the mapping is a bit more complex if the negative

search returns a sibling of the current element. If the node in question is not hanging, then we can copy its value according to the above mentioned mapping. However, if the node in question is hanging, then instead of the mapping, the corresponding node indices from element a are copied. This case is explained in Figure 6(b), where we observe that if node a_1, b_0 is hanging, we need to use $b_0 = a_0$ and use $b_0 = a_1$ if it is a node.

4.2 Mesh Compression

One of the major problems with unstructured meshes is the storage overhead. In the case of the octree, this amounts to having to store both the octree and the lookup table. In order to reduce the storage costs associated with the octree, we compress both the octree and the lookup table. The sorted, unique, linear octree can be easily compressed by retaining only the offset of the first element and the level of subsequent octants. Storing the offset for each octant requires a storage of three integers (12 `byte`s) and a `byte` for storing the level. Storing only the level represents a 12x compression as opposed to storing the offset for every octant.

It is much harder to compress the element to node mapping, which requires eight integers for each element. In order to devise the best compression scheme, we first estimate the distribution of the node indices. The following lemma helps us analyze the distribution of the indices of the nodes of a given element.

LEMMA 1. *The Morton ids of the nodes of a given ele-*
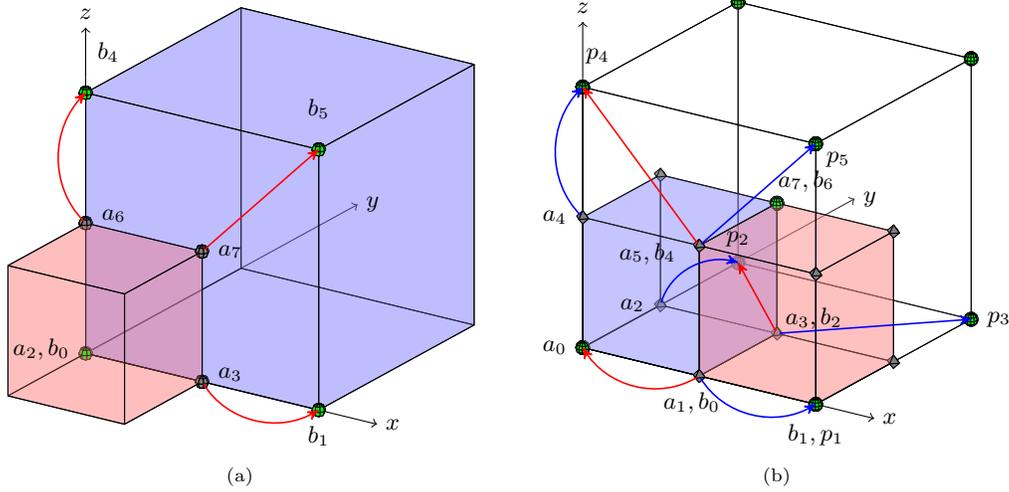
Figure 6: Computing element to node mapping using negative searches. (a) If the found octant (a) is not a sibling of the current octant (b), then the element to node mapping can be copied via the mapping $b_0 \leftarrow a_2$, $b_1 \leftarrow a_3$, $b_4 \leftarrow a_6$, and $b_5 \leftarrow a_7$. (b) In case the found octant (a) is a sibling of the current octant (b), then the mapping depends on whether or not the node in question is hanging. If the node is not hanging, then the same mapping as used in (a) can be applied. If the node is hanging, then the corresponding node indices for the found element are directly copied. For the case shown, $(b_0, b_2, b_4, b_6) \leftarrow (a_0, a_2, a_4, a_7) = (p_0, p_2, p_4, a_7)$.

ment are greater than or equal to the Morton id of the element.

PROOF. Let the anchor of the given element be (x, y, z) and let its size be h. In that case the anchors of the 8 nodes of the element are given by $(x, y, z), (x+h, y, z), (x, y+h, z), (x+h, y+h, z) \cdots$. By the definition of the Morton ordering all of these except (x, y, z) are greater than the Morton id of the element. The node at (x, y, z) is equal to the Morton id of the element. $\square$

COROLLARY 1. *Given a sorted list of Morton ids corresponding to the combined list of elements and nodes of a balanced linear octree, the indices of the 8 nodes of a given element in this list are strictly greater than the index of the element. Moreover, if the nodes are listed in the Morton order, the list of indices is monotonically increasing. If we store offsets in the sorted list, then these offsets are strictly positive.*

Based on these observations we can estimate the expected range of offsets. Let us assume a certain balanced octree, O, with n octants (elements and hanging-nodes) and with maximum possible depth $D_{\max}$. Consider an element in the octree, o_i, whose index is i, $0 \leqslant i < n$. The offset of the anchor of this element is either i (if the anchor is not hanging) or $n_0 < i$. The indices for the remaining 7 nodes do not depend on octants with index less than i. In addition since the indices of the 7 nodes are monotonically increasing, we can store offsets between two consecutive nodes. That is, if the indices of the 8 nodes of an element, o_i, are $(n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7)$, we only need to store $(n_0 - i, n_1 - n_0, n_2 - n_1, n_3 - n_2, n_4 - n_3, n_5 - n_4, n_6 - n_5, n_7 - n_6)$.

To efficiently store these offsets, we need to estimate how large these offsets can be. We start with a regular grid, i.e., a balanced octree with all octants at $D_{\max}$. Note that any octree that can be generated at the same value of $D_{\max}$ can be obtained by applying a series of local coarsening operations to the regular grid. Since we only store the offsets it is sufficient to analyze the distribution of the offset values for one given direction, say for a neighbor along the x-axis. The expression for all other directions are similar.

For $D_{\max} = 0$, there is only one octant and correspondingly the offset is 1. If we introduce a split in the root octant, $D_{\max}$ becomes 1, the offset increases by 2 for one octant. On introducing further splits, the offset is going to increase for those octants that lie on the boundaries of the original splits, and the general expression for the maximum offset can be written as offset $= 1 + \sum_{i=1}^{D_{\max}} 2^{d \cdot i - 1}$, for a d-tree. In addition, a number of other large offsets are produced for intermediate split boundaries. Specifically for a regular grid at maximum depth $D_{\max}$, we shall have $2^{d \cdot (D_{\max} - x)}$ octants with an offset of $1 + \sum_{i=1}^{x} 2^{d \cdot i - 1}$. As can be clearly seen from the expression, the distribution of the offsets is geometric. With the largest number of octants having small offsets.

For the case of general balanced octrees, we observe that any of these can be obtained from a regular grid by a number of coarsening operations. The only concern is whether the coarsening can increase the offset for a given octant. The coarsening does not affect octants that are greater than the current octant (in the Morton order). For those which are smaller, the effect is minimal since every coarsening operation reduces the offsets that need to be stored.

Golomb-Rice coding [9, 18] is a form of entropy encoding that is optimal for geometric distributions, that is, when

small values are vastly more common than large values. Since, the distribution of the offsets is geometric, we expect a lot of offsets with small values and fewer occurrences of large offsets. The Golomb coding uses a tunable parameter M to divide an input value into two parts: q, the result of a division by M, and r, the remainder. In our implementation, the remainder is stored as a `byte`, and the quotient as a `short`. On an average, we observe one large jump in the node indices, and therefore the amortized cost of storing the compressed lookup table, is 8 `bytes` for storing the remainders, 2 `bytes` for the quotient, one `byte` for storing a flag to determine which of the 8 nodes need to use a quotient, and one additional `byte` for storing additional element specific flags. Storing the lookup explicitly would require 8 `ints`, and therefore we obtain a 3x compression in storing the lookup table.

5. FINITE ELEMENT COMPUTATION ON OCTREES

In this section, we describe the evaluation of a `MatVec` with the global finite element 'stiffness' matrix. The `MatVec` refers to a function that takes a vector and returns another vector, the result of applying the discretized PDE operator to the the input vector. A key difference between our `MatVec` and earlier approaches [22] is that the *"hanging nodes[9]"* are not stored explicitly. A method to eliminate hanging nodes in locally refined quadrilateral meshes and yet ensure inter-element continuity by the use of special bilinear quadrilateral elements was presented in [24]. We have extended that approach to three dimensions. All the stencils were derived for the reference element shown in Figure 5(a). We reduce the number of possible stencils using the following property: No octant can have more than 6 hanging nodes and an octant can have a face hanging node only if the remaining nodes on that face are one of the following: (a) edge hanging nodes, (b) the node common to both this octant and its parent, and (c) the node common to this octant and all its siblings. A direct extension of the approach used in [24] to three dimensions along with the use of the above properties will result in 18 different configurations for the reference element. However, these 18 different types can be obtained from eight different types by simple linear transformations. Any other element in the mesh can also be obtained using a simple rotation and scaling of the reference element. Hence, we only store eight element matrices and the integration over any element in the mesh can be performed by mapping the element in question to the reference element and using one of the eight stored matrices.

The `MatVec` can be evaluated either by looping through the elements or by looping through the nodes. Due to the simplicity of the elemental loop, we have chosen to support elemental loops over nodal loops. We first loop over the elements in the interior of the processor domain since these elements do not share any nodes with the neighboring processors. While we perform this computation we communicate the values of the ghost nodes in the background. At the end of the first loop, we use the ghost values from other processors and loop over the remaining elements.

---

[9]Hanging nodes are nodes that exist at the center of a face or an edge of some octant. They do not represent independent degrees of freedom in a FEM solution.

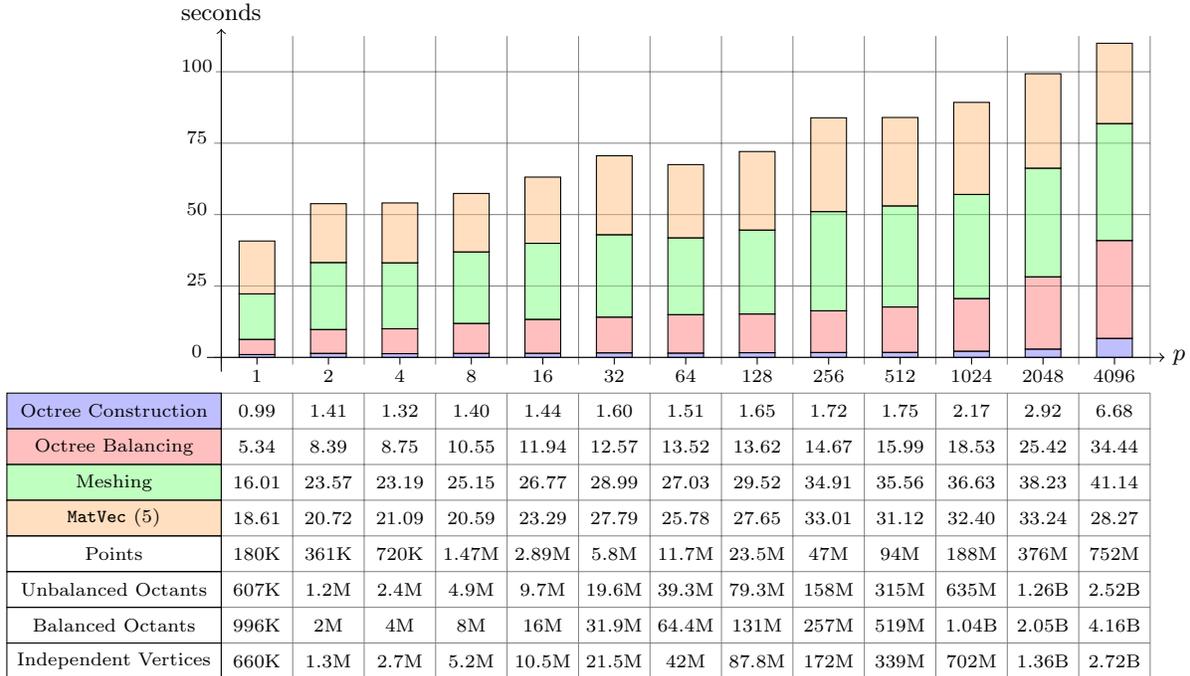| Problem Size	Regular Grid MatVec	Octree Mesh			
		Uniform		Gaussian	
		Meshing	MatVec	Meshing	MatVec
256K	1.08	4.07	1.62	4.34	1.57
512K	2.11	8.48	3.18	8.92	3.09
1M	4.11	17.52	6.24	17.78	6.08
2M	8.61	36.27	11.13	37.29	12.33
4M	17.22	73.74	24.12	76.25	24.22

Table 1: The time to construct (Meshing) and perform 5 matrix-vector multiplications (MatVec) on a single processor for increasing problem sizes. Results are presented for Gaussian distribution and for uniformly spaced points. We compare with matrix-vector multiplication on a regular grid (no indexing) having the same number of elements and the same discretization (trilinear elements). We discretize a variable coefficient (isotropic) operator. All wall-clock times are in seconds. The runs took place on a 2.2 GHz, 32-bit Xeon box. The sustained performance is approximately 400 MFlops/sec for the structured grid. For the uniform and Gaussian distribution of points, the sustained performance is approximately 280 MFlops/sec.

6. PERFORMANCE EVALUATION

In this section we present numerical results for the tree construction, balancing, meshing and matrix vector multiplication for a number of different cases. The algorithms were implemented in C++ using the MPI library. PETSc [4] was used for profiling the code. We consider two point distribution cases: a regular grid one, to directly compare with structured grids; and a Gaussian distribution which resembles a generic non-uniform distribution. In all examples we discretized a variable-coefficient linear elliptic operator. We used piecewise constant coefficients for both the Laplacian and Identity operators. Material properties for the element were stored in an independent array, rather than within the octree data structure.

First we test the performance of the code on a sequential machine. We compare with a regular grid implementation with direct indexing (the nodes are ordered in lexicographic order along the coordinates). The results are presented in Table 1. We report construction times, and the total time for 5 matrix vector multiplications. Overall the code performs quite well. Both the meshing time and the time for performing the `MatVecs` are not sensitive to the distribution. The `MatVec` time is only 50% more than that for a regular grid with direct indexing, about five seconds for four million octants.

In the second set of experiments we test the isogranular scalability of our code. Again, we consider two point distributions, a uniform one and a Gaussian. The size of the input points, the corresponding linear and balanced octrees, the number of vertices, and the runtimes for the two distributions are reported in Figures 7 and 8. All the runs took place on a Cray XT3 MPP system equipped with 2068 compute nodes (two 2.6 GHz AMD Opteron and 2 GBytes of RAM per node) at the Pittsburgh Supercomputing Center. We observe excellent scalability for the construction and balancing of octrees, meshing and the matrix-vector multi-

seconds

	1	2	4	8	16	32	64	128	256	512	1024	2048	4096
Octree Construction	0.99	1.41	1.32	1.40	1.44	1.60	1.51	1.65	1.72	1.75	2.17	2.92	6.68
Octree Balancing	5.34	8.39	8.75	10.55	11.94	12.57	13.52	13.62	14.67	15.99	18.53	25.42	34.44
Meshing	16.01	23.57	23.19	25.15	26.77	28.99	27.03	29.52	34.91	35.56	36.63	38.23	41.14
MatVec (5)	18.61	20.72	21.09	20.59	23.29	27.79	25.78	27.65	33.01	31.12	32.40	33.24	28.27
Points	180K	361K	720K	1.47M	2.89M	5.8M	11.7M	23.5M	47M	94M	188M	376M	752M
Unbalanced Octants	607K	1.2M	2.4M	4.9M	9.7M	19.6M	39.3M	79.3M	158M	315M	635M	1.26B	2.52B
Balanced Octants	996K	2M	4M	8M	16M	31.9M	64.4M	131M	257M	519M	1.04B	2.05B	4.16B
Independent Vertices	660K	1.3M	2.7M	5.2M	10.5M	21.5M	42M	87.8M	172M	339M	702M	1.36B	2.72B

Figure 7: Isogranular scalability for Gaussian distribution of 1M octants per processor. From left to right, the bars indicate the time taken for the different components of our algorithms for increasing processor counts. The bar for each processor is partitioned into 4 sections. From top to bottom, the sections represent the time taken for (1) performing 5 Matrix-Vector multiplications, (2) Construction of the octree based mesh, (3) balancing the octree and (4) construction from points .

plication operation. For example, in Figure 7 we observe the expected complexity in the construction and balancing of the octree (there is a slight growth due to the logarithmic factor in the complexity estimate) and we observe a roughly size-independent behavior for the matrix-vector multiplication.The results are even better for the uniform distribution of points in Figure 8, where the time for 5 matrix-vector multiplications remains nearly constant at approximately 20 seconds.

Finally, we compare the meshing time for the two search strategies presented in Section 4.1. The improvement in meshing time as a result of using the 4-way search is shown in Figure 9, for the Gaussian distribution. As can be seen, there is a significant improvement in meshing time at all processor counts.

7. CONCLUSIONS AND FUTURE WORK
We have presented a set of algorithms for the parallel construction, balancing, and meshing of linear octrees. Our mesh data structure is interfaced with PETSc [4], thus allowing us to use its linear and non-linear solvers. Our target applications include elliptic, parabolic, and hyperbolic partial differential equations. We presented results that verify the overall scalability of our code. The overall meshing time is in the order of one minute for problems with up to four billion elements. Thus, our scheme enables efficient execution of applications which require frequent remeshing.

There are two important extensions: multigrid schemes, and higher-order discretizations. For the former, restriction and prolongation operators need to be designed, along with refinement and coarsening schemes. Higher-order schemes will require additional bookkeeping and longer lookup tables as the inter-element connectivity will increase.

8. ACKNOWLEDGMENTS

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

[2] Volkan Akcelik, Jacobo Bielak, George Biros, Ioannis Epanomeritakis, Antonio Fernandez, Omar Ghattas, Eui Joong Kim, Julio Lopez, David R. O'Hallaron, Tiankai Tu, and John Urbanic. High resolution forward and inverse earthquake modeling on terascale

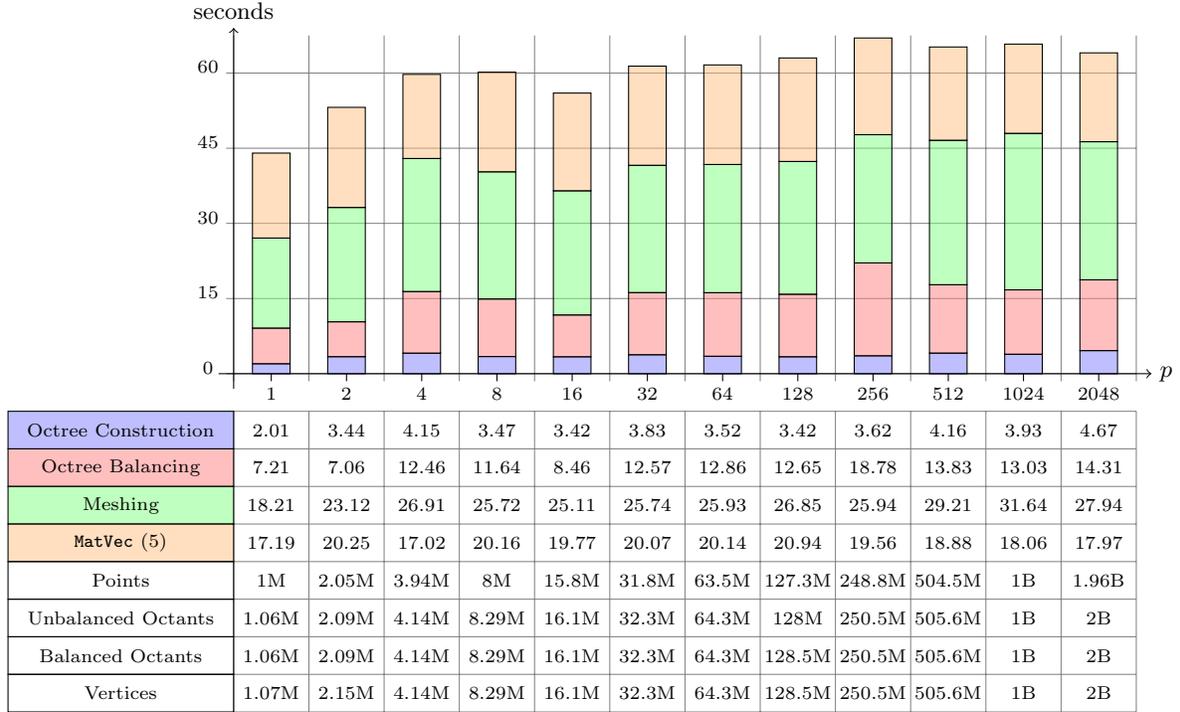|	1	2	4	8	16	32	64	128	256	512	1024	2048
Octree Construction	2.01	3.44	4.15	3.47	3.42	3.83	3.52	3.42	3.62	4.16	3.93	4.67
Octree Balancing	7.21	7.06	12.46	11.64	8.46	12.57	12.86	12.65	18.78	13.83	13.03	14.31
Meshing	18.21	23.12	26.91	25.72	25.11	25.74	25.93	26.85	25.94	29.21	31.64	27.94
MatVec (5)	17.19	20.25	17.02	20.16	19.77	20.07	20.14	20.94	19.56	18.88	18.06	17.97
Points	1M	2.05M	3.94M	8M	15.8M	31.8M	63.5M	127.3M	248.8M	504.5M	1B	1.96B
Unbalanced Octants	1.06M	2.09M	4.14M	8.29M	16.1M	32.3M	64.3M	128M	250.5M	505.6M	1B	2B
Balanced Octants	1.06M	2.09M	4.14M	8.29M	16.1M	32.3M	64.3M	128.5M	250.5M	505.6M	1B	2B
Vertices	1.07M	2.15M	4.14M	8.29M	16.1M	32.3M	64.3M	128.5M	250.5M	505.6M	1B	2B

Figure 8: Isogranular scalability for uniformly spaced points with 1M octants per processor. From left to right, the bars indicate the time taken for the different components of our algorithms for increasing processor counts. The bar for each processor is partitioned into 4 sections. From top to bottom, the sections represent the time taken for (1) performing 5 Matrix-Vector multiplications, (2) Construction of the octree based mesh, (3) balancing the octree and (4) construction from points.

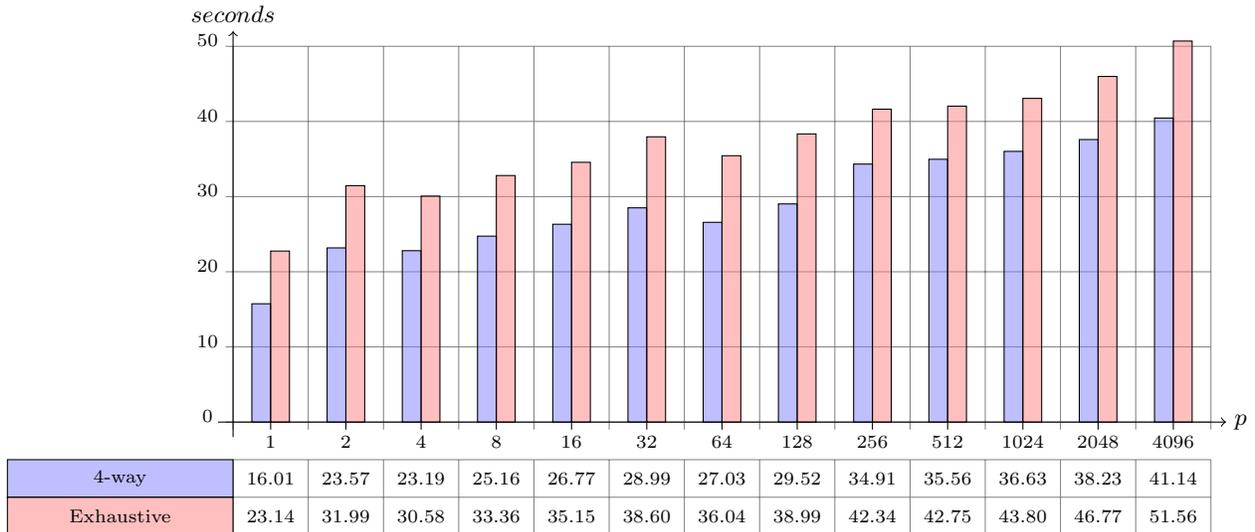|	1	2	4	8	16	32	64	128	256	512	1024	2048	4096
4-way	16.01	23.57	23.19	25.16	26.77	28.99	27.03	29.52	34.91	35.56	36.63	38.23	41.14
Exhaustive	23.14	31.99	30.58	33.36	35.15	38.60	36.04	38.99	42.34	42.75	43.80	46.77	51.56

Figure 9: Comparison of meshing times using exhaustive search with using a hybrid approach where only the first layer of octants uses exhaustive search and the rest use the 4-way search to construct the lookup tables. The test was performed using a Gaussian distribution of 1 million octants per processor. It can be seen that the 4-way search is faster than the exhaustive search and scales upto 4096 processors.

computers. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. ACM, 2003.

[3] WK Anderson, WD Gropp, DK Kaushik, DE Keyes, and BF Smith. Achieving high sustained performance in an unstructured mesh CFD application. *Supercomputing, ACM/IEEE 1999 Conference*, pages 69–69, 1999.

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

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

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

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

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

[9] S.W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.

[10] Leslie Greengard and William Gropp. A parallel version of the fast multipole method-invited talk. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 213–222, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

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

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

[13] Bhanu Hariharan, Srinivas Aluru, and Balasubramaniam Shanker. A scalable parallel fast multipole method for analysis of scattering from perfect electrically conducting surfaces. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

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

[15] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Comput. Surv.*, 19(3):261–296, 1987.

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

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

[18] R. F. Rice. Some practical universal noiseless coding techniques. Technical Report JPL Publication 79-22, Jet Propulsion Laboratory, Pasadena, California, 1979.

[19] Hari Sundar, Rahul S. Sampath, and George Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *University of Pennsylvania Technical Report*, MS-CIS-07-05, 2007.

[20] Tiankai Tu and David R. O'Hallaron. Balance refinement of massive linear octree datasets. *CMU Technical Report*, CMU-CS-04(129), 2004.

[21] Tiankai Tu and David R. O'Hallaron. Extracting hexahedral mesh structures from balanced linear octrees. In *13th International Meshing Roundtable*, pages 191–200, Williamsburg, VA, Sandia National Laboratories, September 19-22 2004.

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

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

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

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

[26] Lexing Ying, George Biros, and Denis Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *J. Comput. Phys.*, 196(2):591–626, 2004.

[27] Lexing Ying, George Biros, Denis Zorin, and Harper Langston. A new parallel kernel-independent fast multipole method. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 14, Washington, DC, USA, 2003. IEEE Computer Society.