SIAM J. Sci. Comput.
Vol. 30, No. 5, pp. 2675–2708

# BOTTOM-UP CONSTRUCTION AND 2:1 BALANCE REFINEMENT OF LINEAR OCTREES IN PARALLEL*

HARI SUNDAR[†], RAHUL S. SAMPATH[‡], AND GEORGE BIROS[§]

**Abstract.** In this article, we propose new parallel algorithms for the construction and 2:1 balance refinement of large linear octrees on distributed memory machines. Such octrees are used in many problems in computational science and engineering, e.g., object representation, image analysis, unstructured meshing, finite elements, adaptive mesh refinement, and N-body simulations. Fixed-size scalability and isogranular analysis of the algorithms using an MPI-based parallel implementation was performed on a variety of input data and demonstrated good scalability for different processor counts (1 to 1024 processors) on the Pittsburgh Supercomputing Center's TCS-1 AlphaServer. The results are consistent for different data distributions. Octrees with over a billion octants were constructed and balanced in less than a minute on 1024 processors. Like other existing algorithms for constructing and balancing octrees, our algorithms have $\mathcal{O}(N \log N)$ work and $\mathcal{O}(N)$ storage complexity. Under reasonable assumptions on the distribution of octants and the work per octant, the parallel time complexity is $\mathcal{O}(\frac{N}{n_p} \log(\frac{N}{n_p}) + n_p \log n_p)$, where $N$ is the size of the final linear octree and $n_p$ is the number of processors.

**Key words.** linear octrees, balance refinement, Morton encoding, large scale parallel computing, space filling curves

**AMS subject classifications.** 65N50, 65Y05, 68W10, 68W15

**DOI.** 10.1137/070681727

## 1. Introduction.

Spatial decompositions of the $d$-dimensional cube have important applications in scientific computing: they can be used as algorithmic foundations for adaptive finite element methods [3, 17], adaptive mesh refinement methods [14, 22], and many-body algorithms [15, 30, 35, 37, 38]. *Quadtrees* [9] and *octrees* [19] are hierarchical data structures commonly used for partitioning 2- and 3-dimensional domains, respectively; they use axis-aligned lines and planes, respectively. These tree data structures have been in use for over three decades now [9, 23]. However, design and use of large scale distributed tree data structures that scale to thousands of processors is still a major challenge and is an area of active research even today [4, 7, 10, 14, 15, 30, 34, 35, 36, 37, 38].

Octrees and quadtrees are usually employed while solving the following two types of problems.

- *Searching:* Searches within a domain using $d$-trees ($d$-dimensional trees with a maximum of $2^d$ children per node) benefit from the reduction of the com-

*Received by the editors February 3, 2007; accepted for publication (in revised form) April 4, 2008; published electronically August 6, 2008. This work was supported by the U.S. Department of Energy under grant DE-FG02-04ER25646 and by the U.S. National Science Foundation under grants CCF-0427985, CNS-0540372, and DMS-0612578. Computing resources on the TeraGrid's HP AlphaCluster system at the Pittsburgh Supercomputing Center were provided under the MCA04T026 award.

http://www.siam.org/journals/sisc/30-5/68172.html

[†] Department of Bioengineering, University of Pennsylvania, 120 Hayden Hall, 3320 Smith Walk, Philadelphia, PA 19104-2688 (hsundar@seas.upenn.edu).

[‡] Department of Mechanical Engineering and Applied Mechanics, University of Pennsylvania, 220 S. 33rd Street, Philadelphia, PA 19104-6315 (rahulss@seas.upenn.edu).

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

plexity of the search from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$ [11, 21].

- *Spatial decomposition:* Unstructured meshes are often preferred over uniform discretizations because they can be used with complicated domains and permit rapid grading from coarse to fine elements. However, generating large unstructured meshes is a challenging task [27]. On the contrary, octree-based unstructured hexahedral meshes can be constructed efficiently [5, 13, 24, 25, 26, 33]. Although they are not suitable for highly complicated geometries, they provide a good compromise between adaptivity and simplicity for numerous applications like solid modeling [19], object representation [1, 6], visualization [10], image segmentation [29], adaptive mesh refinement [14, 22], and N-body simulations [15, 30, 35, 37, 38].

Octree data structures used in discretizations of partial differential equations should satisfy certain spatial distribution of octant size [4, 34]. That is, there is a restriction on the relative sizes of adjacent octants.[1] Furthermore, conforming discretizations require the "balance condition" to construct appropriate function spaces. In particular, when the 2:1 balance constraint is imposed on octree-based hexahedral meshes, it ensures that there is at most one *hanging* node on any edge or face. What makes the balance-refinement problem difficult and interesting is a property known as the *ripple effect*: An octant can trigger a sequence of splits whereby it can force an octant to split, even if it is not in its immediate neighborhood. Hence, balance refinement is a nonlocal and inherently iterative process. Solving the balance-refinement problem in parallel introduces further challenges in terms of synchronization and communication since the ripple can propagate across multiple processors.

*Related work.* Limited work has been done on large scale parallel construction [15, 36, 38] and balance refinement [17, 34] of octrees, and the best known algorithms exhibit suboptimal isogranular scalability. The key component in constructing octrees is the partitioning of the input in order to achieve good load balancing. The use of space-filling curves for partitioning data has been quite popular [15, 34, 36, 38]. The proximity preserving property of space-filling curves makes them attractive for data partitioning. All of the existing algorithms for constructing octrees use a top-down approach after the initial partition. The major hurdle in using a parallel top-down approach is avoiding overlaps. This typically requires some synchronization after constructing a portion of the tree [34, 36, 38]. Section 3.2 describes the issues that arise in using a parallel top-down approach.

Bern, Eppstein, and Teng [4] proposed an algorithm for constructing and balancing quadtrees for EREW PRAM architectures. However, it cannot be easily adapted for distributed architectures. In addition, the balanced quadtree produced is suboptimal and can have up to 4 times as many cells as the optimal balanced quadtree. Tu, O'Hallaron, and Ghattas [34] proposed a more promising approach, which was evaluated on large octrees. They constructed and balanced 1.22B octants for the Greater Los Angeles basin dataset [18] on 2000 processors in about 300 seconds. This experiment was performed on the TCS-1 terascale computing HP AlphaServer Cluster at the Pittsburgh Supercomputing Center. In contrast, we construct and balance[2] 1B octants (approximately) for three different point distributions (Gaussian, log-normal, and uniform) on 1024 processors on the same cluster in about 60 seconds.

[1] This is referred to as the balance constraint. A formal definition of this constraint is given in section 2.2.

[2] While we enforce the 0-*balance* constraint, [34] enforces only the 1-*balance* constraint. Note that it is harder to 0-*balance* a given octree. See section 2.2 for more details on the different balance constraints.

*Synopsis and contributions.* In this paper we present two parallel algorithms: one to construct complete linear octrees from a list of points, and one to enforce an optimal 2:1 balance constraint[3] on complete linear octrees. We use a linear octree Morton encoding–based representation. Given a set of points, partitioned across processors, we create a set of octants that we sort and repartition using the Morton ordering. A complete linear octree is constructed using the seed octants. Then we build an additional auxiliary list of a small number of coarse octants or *blocks.* This auxiliary octant set encapsulates and compresses the local spatial distribution of octants; it is used to accelerate the 2:1 balance refinement, which we implement using a hybrid strategy: intrablock balancing is performed by a classical level-by-level balancing/duplicate-removal scheme; and interblock balancing is performed by a variant of the ripple-propagation algorithm proposed in [34]. The main parallel tools used are sample sorts (accelerated by biotonic sorts) and standard point-to-point/collective communication calls.[4]

In a nutshell, the major contributions of this work are as follows:

- A parallel bottom-up algorithm for coarsening octrees, which is also used for partitioning the input in our other algorithms.
- A parallel bottom-up algorithm for constructing linear octrees. We avoid the synchronization issues that are usually associated with parallel top-down methods.
- An algorithm for enforcing 2:1 balance refinement in parallel. The algorithm constructs the minimum number of nodes to satisfy the 2:1 constraint. Its key feature is that it avoids parallel searches, which, as we show in sections 3.3.6 and 3.3.7, are the main hurdles in achieving good isogranular scalability.

*Remark.* The main parallel cost of the algorithm is that related to the parallel sorts that run in $\mathcal{O}(N \log N)$ work and $\mathcal{O}(\frac{N}{n_p} \log(\frac{N}{n_p}) + n_p \log(n_p))$ time, assuming uniformly distributed points [12]. In the following sections we present several algorithms for which we give precise work and storage complexity. For some of the parallel algorithms we also give time complexity estimates; this corresponds to wall-clock time and includes work per processor and communication costs. The precise number depends on the initial distribution and the effectiveness of the partitioning. Thus the numbers for time are only an estimate under uniform distribution assumptions. If the time complexity is not specifically mentioned, then it is comparable to that of a sample sort.

*Organization of the paper.* In section 2 we introduce some terminology that will be used in the rest of the paper. In section 3, we describe the various components of our construction and balance refinement algorithms. In section 4, we present numerical experiments, including fixed size and isogranular scalability tests on different data distributions. Finally, in section 5, shortcomings of the proposed approach are discussed and some suggestions for future work are also offered. Tables 1 and 2 summarize the notation that is used in the subsequent sections.

## 2. Background.

An octree is a tree data structure in which every node has a maximum of eight children. Octrees are analogous to binary trees (maximum of two children per node) in one dimension and quadtrees (maximum of four children per node) in two dimensions. A node with no children is called a *leaf* and a node with one

[3] There exists a unique least common balance refinement for a given octree [20].

[4] When we discuss communication costs, we assume a hypercube network topology with $\Theta(n_p)$ bisection width.

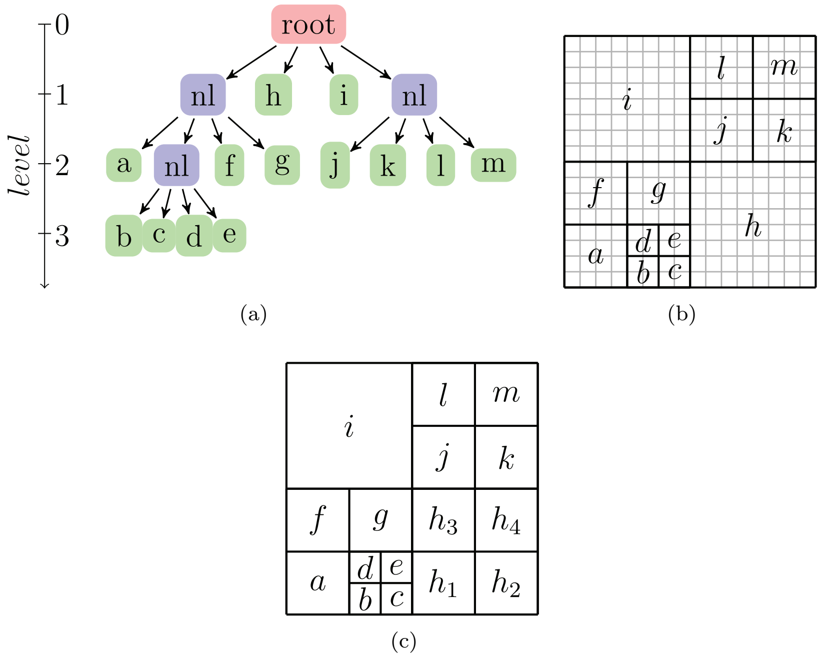
TABLE 1
*Symbols for terms.*

| | |
|---|---|
| $\mathcal{L}(N)$ | Level of octant $N$. |
| $\mathcal{L}^*$ | Maximum level attained by any octant. |
| $D_{max}$ | Maximum permissible depth of the tree (upper bound for $\mathcal{L}^*$). |
| $\mathcal{P}(N)$ | Parent of octant $N$. |
| $\mathcal{B}(N)$ | The block that is equal to or is an ancestor of octant $N$. |
| $\mathcal{S}(N)$ | Siblings (sorted) of octant $N$. |
| $\mathcal{C}(N)$ | Children (sorted) of octant $N$. |
| $\mathcal{D}(N)$ | Descendant of octant $N$. |
| $\mathcal{FC}(N)$ | First child of octant $N$. |
| $\mathcal{LC}(N)$ | Last child of octant $N$. |
| $\mathcal{FD}\,(N,l)$ | First descendant of octant $N$ at level $l$. |
| $\mathcal{LD}\,(N,l)$ | Last descendant of octant $N$ at level $l$. |
| $\mathcal{DFD}(N)$ | Deepest first descendant of octant $N$. |
| $\mathcal{DLD}(N)$ | Deepest last descendant of octant $N$. |
| $\mathcal{A}(N)$ | Ancestor of octant $N$. |
| $\mathcal{A}_{finest}\,(N,K)$ | Nearest common ancestor of octants $N$ and $K$. |
| $\mathcal{N}\,(N,l)$ | List of all potential neighbors of octant $N$ at level $l$. |
| $\mathcal{N}^s\,(N,l)$ | A subset of $\mathcal{N}(N,l)$, with the property that all of these share the same common corner with $N$. This is also the corner that $N$ shares with its parent. |
| $\mathcal{N}\,(N)$ | Neighbor of $N$ at any level. |
| $\mathcal{I}(N)$ | Insulation layer around octant $N$. |
| $N^p_{max}$ | Maximum number of points per octant. |
| $n_p$ | Total number of processors. |
| $A_{global}$ | Union of the list $A$ from all the processors. |
| $\{\cdots\}$ | A set of elements. |
| $\emptyset$ | The empty set. |

TABLE 2
*Symbols for operations.*

| | |
|---|---|
| $A \leftarrow B$ | Assignment operation. |
| $A \oplus B$ | Bitwise $A$ `XOR` $B$. |
| $\{A\} \cup \{B\}$ | Union of the sets $A$ and $B$. The order is preserved, if possible. |
| $\{A\} \cap \{B\}$ | Intersection of the sets $A$ and $B$. |
| $A + B$ | The list formed by concatenating the lists $A$ and $B$. |
| $A - B$ | Remove the contents of $B$ from $A$. |
| $A[i]$ | $i$th element in list $A$. |
| `len`($A$) | Number of elements in list $A$. |
| `Sort`($A$) | Sort $A$ in the ascending Morton order. |
| $A$.`push_front`($B$) | Insert $B$ to the beginning of $A$. |
| $A$.`push_back`($B$) | Append $B$ to the end of $A$. |
| `Send`($A$ ,$r$ ) | Send $A$ to processor with rank $= r$. |
| `Receive()` | Receive from any processor. |

or more children is called an *interior node*. 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*. A node's children, grandchildren, and so on are collectively referred to as the node's *descendants*, and this node will be an *ancestor* of its descendants. A node along with all its descendants can be viewed as a separate tree in itself with this node as its root. Hence, this set is also referred to as a *subtree* of the original tree. The depth of a node 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 every interior node is one level lower than its children.



(a) (b) (c)

FIG. 1. (a) *Tree representation of a quadtree;* (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.*

Octrees and quadtrees[5] 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. Alternatively, one can also define complete octrees as octrees in which every interior node has exactly eight child nodes. We will frequently use the equivalence of these two definitions.

There are many different ways to represent trees [8]. In this work, we will use a linearized representation of octrees known as *linear octrees*. In this representation, we discard the interior nodes and only store the complete list of leaves. This representation is advantageous for the following reasons.

- It has lower storage costs than other representations.
- The other representations use pointers, which add synchronization and communication overhead for parallel implementations.

To use a linear representation, a *locational 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. The following section describes one such locational code known as the *Morton encoding*.[6]

[5] All the algorithms described in this paper are applicable to both octrees and quadtrees. For simplicity, we will use quadtrees to illustrate the concepts in this paper and use the terms "octrees" and "octants," consistently, in the rest of the paper.

[6] Morton encoding is one of many space-filling curves [7]. Our algorithms are generic enough to work with other space-filling curves as well. However, Morton encoding is relatively simpler to implement since, unlike other space-filling curves, no rotations or reflections are performed.

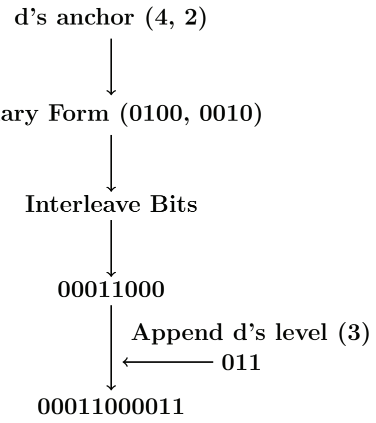


Fig. 2. *Computing the Morton id of quadrant "d" in the quadtree shown in Figure* 1(b). *The anchor for any quadrant is its lower left corner.*

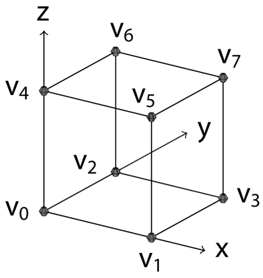


Fig. 3. *Orientation for an octant. By convention,* $v_0$ *is chosen as the anchor of the octant. The vertices are numbered in the Morton ordering.*

**2.1. Morton encoding.** In order to construct a Morton encoding, the maximum permissible depth, $D_{max}$, of the tree is specified a priori. Note that $D_{max}$ is different from $\mathcal{L}^*$, the maximum level attained by any node. In general, $\mathcal{L}^*$ cannot be specified a priori. $D_{max}$ is only a weak upper bound for $\mathcal{L}^*$.

The domain is represented by a uniform grid of $2^{D_{max}}$ indivisible cells in each dimension (Figure 1(b)). 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 (Figure 2). By convention, the anchor of a quadrant is its lower left corner and the anchor of an octant is its lower left corner facing the reader (corner $v_0$ in Figure 3).

The Morton encoding for any octant is derived by interleaving[7] 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 [4, 7, 31, 34]. Interesting properties of the Morton encoding scheme are listed in Appendix A. In the rest of the paper the terms *lesser* and *greater* and the symbols < and > are used to compare octants based on their Morton ids (i.e.,

[7] Instead of bit-interleaving as described here, we use a multicomponent version (Appendix B) of the Morton encoding scheme.

identification), and *coarser* and *finer* to compare them based on their relative sizes, i.e., their levels in the octree.

**2.2. Balance constraint.** In many applications involving octrees, it is desirable to impose a restriction on the relative sizes of adjacent octants [16, 17, 34]. Generalizing Moore's [20] categorization of the general balance conditions, we have the following definition for the 2:1 balance constraint.

DEFINITION 1. *A linear d-tree is k-balanced if and only if, for any $l \in [1, \mathcal{L}^*)$, no leaf at level $l$ shares an m-dimensional face*[8] $(m \in [k, d))$ *with another leaf at level greater than $l + 1$.*

For the specific case of octrees we use 2-*balanced* to refer to octrees that are balanced across faces, 1-*balanced* to refer to octrees that are balanced across edges and faces, and 0-*balanced* to refer to octrees that are balanced across corners, edges, and faces. The result of imposing the 2:1 balance constraint is that no octant can be more than twice as coarse as its adjacent octants. Similarly, 4:1 and higher constraints can be imposed. In this work, we will restrict the discussion to 2:1 balancing alone. However, the algorithms presented in this work can be extended easily to satisfy higher balance constraints as well. An example of a 0-*balanced* quadtree is shown in Figure 1(c). The balance algorithm proposed in this work is capable of *k-balancing* a given complete linear octree, and since it is hardest to 0-*balance* a given octree we report all results for the 0-*balance* case.

**3. Algorithms.** We will first describe a key algorithmic component (section 3.1) that forms the backbone for both our parallel octree construction and balancing algorithms. This is a partition heuristic known as *block partition* and is specifically designed for octrees. It has two main subcomponents, which are described in sections 3.1.1 and 3.1.2.

We then present the parallel octree construction algorithm in section 3.2 and the parallel balancing algorithm in section 3.3. The overall parallel balancing algorithm (Algorithm 11) is made up of numerous components, which are described in sections 3.3.1 through 3.3.6.

**3.1. Block partition.** A simple way to partition the domain into a union of blocks would be to take a top-down approach and create a coarse regular grid, which can be divided[9] among the processors. However, this approach does not take load balancing into account since it does not use the underlying data distribution. Alternatively, one could use a space-filling curve to sort the octants and then partition them so that every processor gets an almost equal sized chunk of octants, contiguous in this order. This can be done by assigning the same weight to all the octants and then using Algorithm 1. However, this approach does not avoid overlaps.

Two desirable qualities of any partitioning strategy are load balancing and minimization of overlap between the processor domains. We use a novel parallel bottom-up coarsening strategy to achieve these. The main intuition behind this partition algorithm (Algorithm 2) is 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 3 main stages:

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

[9] If we create a regular grid at level $l$, then the number of cells will be $n = 2^{dl}$, where $d$ is the dimension. $l$ is chosen in such a way that $n > p$.

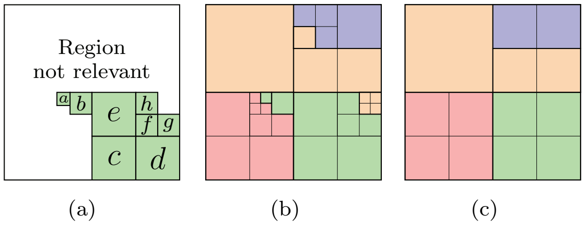


FIG. 4. (a) *A minimal list of quadrants covering the local domain on a processor,* (b) *a Morton ordering–based partition of a quadtree across* 4 *processors, and* (c) *the coarse quadrants and the final partition produced by using the quadtree shown in* (b) *as input to Algorithm* 2.

1. Constructing a distributed coarse complete linear octree that is representative of the underlying data distribution.
2. Assigning weights to the octants in the coarse octree and partitioning them to achieve almost uniform load across the processors.
3. Projecting the partitioning computed in the previous step 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 4(a)) and complete the region between them, as described in section 3.1.1, to obtain a list of coarse octants. We then select the coarsest cell(s) out of this list of coarse octants (octant $e$ in Figure 4(a)). We use the selected octants at each processor and construct a complete linear octree as described in section 3.1.2. The leaves of this complete linear octree are referred to as *blocks*. This gives us a distributed coarse complete linear octree that is based on the underlying data distribution.[10]

We compute the load of each of the blocks created above by computing the number of original octants that lie within it. The blocks are then distributed across the processors using Algorithm 1 so that the total weight on each processor is roughly the same.[11]

The original octants are then partitioned to align with the coarse block boundaries. 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).

Algorithm 2 lists all the steps described above and Figures 4(b) and 4(c) illustrate a sample input to Algorithm 2 and the corresponding output, respectively.

**3.1.1. Constructing a minimal linear octree between two octants.** Given two octants, $a$ and $b > a$, we wish to generate the minimal number of octants that span the region between $a$ and $b$ according to the Morton ordering. The algorithm (Algorithm 3) first calculates 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

[10] Refer to Appendix C for an estimate of the number of blocks produced.

[11] Some of the coarse blocks could be split if it facilitates achieving better load balance across the processors.

Algorithm 1. Partitioning a distributed list of octants (parallel) - `Partition`.

```
Input:     A distributed list of octants, $W$.
Output:    The octants redistributed across processors so that
           the total weight on each processor is roughly the same.
           The relative order of the octants is preserved.
Work:      $\mathcal{O}(n)$, where $n = $ len($W$).
Storage:   $\mathcal{O}(n)$, where $n = $ len($W$).

$S \leftarrow$ Scan( weight($W$) )
if rank = $(n_p - 1)$
   TotalWeight $\leftarrow \max(S)$
   Broadcast(TotalWeight)
end if
$\bar{w} \leftarrow \frac{\text{TotalWeight}}{n_p}$
$k \leftarrow$ (TotalWeight) $\bmod n_p$
$Q_{tot} \leftarrow \emptyset$
for $p \leftarrow 1$ to $n_p$
    if $p \leq k$
       $Q \leftarrow \{x \in W \mid (p-1).(\bar{w}+1) \leq S(x) < p.(\bar{w}+1)\}$
    else
       $Q \leftarrow \{x \in W \mid (p-1).\bar{w}+k \leq S(x) < p.\bar{w}+k\}$
    end if
    $Q_{tot} \leftarrow Q_{tot} + Q$
    Send($Q$, $(p-1)$)
end for
$R \leftarrow$ Receive()
$W \leftarrow W - Q_{tot} + R$
```

Algorithm 2. Partitioning octants into large contiguous blocks (parallel) - `BlockPartition`.

```
Input:     A distributed sorted list of octants, $F$.
Output:    A list of the blocks, $G$.  $F$ is redistributed,
           but the relative order of the octants is preserved.
Work:      $\mathcal{O}(n)$, where $n = $ len($F$).
Storage:   $\mathcal{O}(n)$, where $n = $ len($F$).
Time:      Refer to Appendix C.

$T \leftarrow$ CompleteRegion($F[1]$, $F[$len($F$)$]$) ( Algorithm 3 )
$C \leftarrow \{x \in T \mid \forall y \in T,\ \mathcal{L}(x) \leq \mathcal{L}(y)\}$
$G \leftarrow$ CompleteOctree($C$) ( Algorithm 4 )
for each $g \in G$
    weight($g$) $\leftarrow$ len$\big(F_{global} \cap \{g, \{\mathcal{D}(g)\}\}\big)$
end for
Partition($G$) ( Algorithm 1 )
$F \leftarrow F_{global} \cap \{\{g, \{\mathcal{D}(g)\}\},\ \forall\ g \in G\}$
```

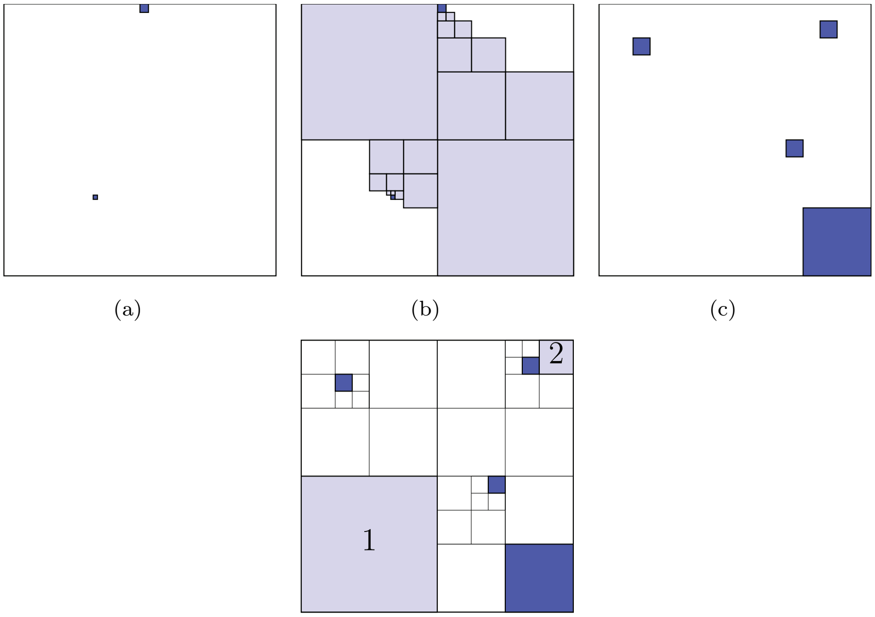
further splits are necessary. This produces the minimal coarse complete linear octree (Figure 5(b)) between the two octants $a$ and $b$ (Figure 5(a)). This algorithm is based on Properties 3 and 4 of the Morton ordering, which are listed in Appendix A.

**3.1.2. 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 5(c)), we use Algorithm 4. The octants are initially sorted based on the Morton ordering. Algorithm 7 is subsequently used to remove overlaps, if any. Two additional octants are added to complete the domain (Figure 5(d)). The first is the

Algorithm 3. Constructing a minimal linear octree between two octants (sequential) - CompleteRegion.

**Input:** Two octants, $a$ and $b > a$.
**Output:** $R$, the minimal linear octree between $a$ and $b$.
**Work:** $\mathcal{O}(n \log n)$, where $n =$ len($R$).
**Storage:** $\mathcal{O}(n)$, where $n =$ len($R$).

```
$W \leftarrow \mathcal{C}(\mathcal{A}_{finest}(a, b))$
for each $w \in W$
    if $(a < w < b)$ AND $(w \notin \{\mathcal{A}(b)\})$
        $R \leftarrow R + w$
    else if $(w \in \{\{\mathcal{A}(a)\}, \{\mathcal{A}(b)\}\})$
        $W \leftarrow W - w + \mathcal{C}(w)$
    end if
end for
Sort(R)
```



(a)

(b)

(c)

(d)

Fig. 5. (b) *The minimal number of octants between the cells given in* (a). *This is produced by using* (a) *as an input to Algorithm* 3. (d) *The coarsest possible complete linear quadtree containing all the cells in* (c). *This is produced by using* (c) *as an input to Algorithm* 4. *The figure also shows the two additional octants added to complete the domain. The first one is the coarsest ancestor of the least possible octant (the deepest first descendant of the root octant), which does not overlap the least octant in the input. This is also the first child of the nearest common ancestor of the least octant in the input and the deepest first descendant of root. The second is the coarsest ancestor of the greatest possible octant (the deepest last descendant of the root octant), which does not overlap the greatest octant in the input. This is also the last child of the nearest common ancestor of the greatest octant in the input and the deepest last descendant of root.*

Algorithm 4. Constructing a complete linear octree from a partial (incomplete) set of octants (parallel) - CompleteOctree.

**Input:** A distributed sorted list of octants, $L$.
**Output:** $R$, the complete linear octree.
**Work:** $\mathcal{O}(n \log n)$, where $n = \texttt{len}(R)$.
**Storage:** $\mathcal{O}(n)$, where $n = \texttt{len}(R)$.

```
RemoveDuplicates($L$)
$L \leftarrow$ Linearise($L$) ( Algorithm 7 )
Partition($L$) ( Algorithm 1 )
if rank = 0
   $L$.push_front($\mathcal{FC}$ ($\mathcal{A}_{finest}$ ($\mathcal{DFD}$( root ), $L[1]$)))
end if
if rank = $(n_p - 1)$
   $L$.push_back($\mathcal{LC}$ ($\mathcal{A}_{finest}$ ($\mathcal{DLD}$( root), $L[\texttt{len}\ (L)]$)))
end if
if rank > 0
   Send($L[1]$ ,(rank$-1$) )
end if
if rank < $(n_p - 1)$
   $L$.push_back( Recieve() )
end if
for $i \leftarrow 1$ to (len($L$) $- 1$)
     $A \leftarrow$ CompleteRegion ($L[i], L[i+1]$) ( Algorithm 3 )
     $R \leftarrow R + L[i] + A$
end for
if rank = $(n_p - 1)$
   $R \leftarrow R + L[\texttt{len}(L)]$
end if
```

coarsest ancestor of the least possible octant (the deepest first descendant of the root octant, Property 7), which does not overlap the least octant in the input. This is also the first child of the nearest common ancestor of the least octant in the input and the deepest first descendant of root. The second is the coarsest ancestor of the greatest possible octant (the deepest last descendant of the root octant, Property 9), which does not overlap the greatest octant in the input. This is also the last child of the nearest common ancestor of the greatest octant in the input and the deepest last decendant of root. The octants are distributed across the processors to get a weight-based uniform load distribution. The local complete linear octree is subsequently generated by completing the region between every consecutive pair of octants as described in section 3.1.1. 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.

**3.2. Constructing large linear octrees in parallel.** Octrees are usually constructed by using a top-down approach: starting with the root octant, cells are split iteratively based on some criteria, until no further splits are required. This is a simple and efficient sequential algorithm. However, its parallel analogue is not so. We use the case of point datasets to discuss some shortcomings of a parallel top-down tree construction. Formally, the problem might be stated as follows: Construct a complete linear octree in parallel from a distributed set of points in a domain with the constraint that no octant should contain more than ($N^p_{max}$) number of points. Each processor can independently construct a tree using a top-down approach on its local set of points. Constructing a global linear octree requires a parallel merge. Merging, Algorithm 5. Constructing a complete linear octree from a distributed list of points (parallel) - Points2Octree.

```
Input:    A distributed list of points, $L$ and a parameter, $(N^p_{max})$,
          which specifies the maximum number of points per octant.
Output:   Complete linear Octree, $B$.
Work:     $\mathcal{O}(n \log n)$, where $n = $ len$(L)$.
Storage:  $\mathcal{O}(n)$, where $n = $ len$(L)$.

$F \leftarrow$ [Octant$(p, D_{max}), \forall p \in L$]
Sort$(F)$
$B \leftarrow$ BlockPartition$(F)$ ( Algorithm 2 )
for each $b \in B$
    if NumberOfPoints$(b)$ $> N^p_{max}$
        $B \leftarrow B - b + \mathcal{C}(b)$
    end if
end for
```

however, is not straightforward.

1. Consider the case where the local number of points in some region on every processor was less than $(N^p_{max})$, and hence all the processors end up having the same level of coarseness in the region. However, the total number of points in that region could be more than $(N^p_{max})$ and hence the corresponding octant should be refined further.
2. In most applications, we would also like to associate a unique processor to each octant. Thus, duplicates across processors must be removed.
3. For linear octrees overlaps across processors must be resolved.
4. Since there might be overlaps and duplicates, not all the work done by the processors can be accounted as useful work. This is a subtle yet important point to consider while analyzing the algorithm for load balancing.

Previous work [15, 34, 36, 38] on this problem has addressed these issues; however, all the existing algorithms involve many synchronization steps and thus suffer from a sizable overhead, resulting in suboptimal isogranular scalability. Instead, we propose a bottom-up approach for constructing octrees from 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.

First, all points are converted into octants at the maximum depth and then partitioned across the processors using the algorithm described in section 3.1. 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.[12] This process is continued until no further splits are required. This procedure is summarized in Algorithm 5.

### 3.3. Balancing large linear octrees in parallel.

Balance refinement is the process of refining (subdividing) nodes in a complete linear octree which fail to satisfy the balance constraint described in section 2.2. The nodes 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

[12] Refer to Appendix D on how to sample the points in order to construct the coarsest possible octree.

process has to be repeated iteratively. The fact that an octant can affect octants not immediately adjacent to it is known as the *ripple effect*.

We use a two-stage balancing scheme: first we perform local balancing on each processor, and follow this up by balancing across the interprocessor boundaries. One of the goals is to get a union of blocks (select nonleaf nodes of the octree) to reside on each processor so that the surface area and thereby the corresponding interprocessor boundaries are minimized. Determining whether a given partition provides the minimal surface area[13] is NP complete, and determining the optimal partition is NP hard, since the problem is equivalent to the set-covering problem [8].

We use the parallel bottom-up coarsening and partitioning algorithm (described in section 3.1) to construct coarse blocks on each processor and to distribute the underlying octants. By construction, the domains covered by these blocks are disjoint and the union of these blocks covers the entire domain. We use the blocks as a means to minimize the number of octants that need to be split due to interprocessor violations of the 2:1 balancing rule.

**3.3.1. Local balancing.** There are two approaches for balancing a complete octree. In the first approach, every node constructs the coarsest possible neighbors satisfying the balance constraint, and subsequently any duplicates and overlaps are removed [4]. We describe this approach in Algorithm 6. In an alternative approach, the nodes search for neighbors and resolve any violations of the balance constraint [32, 34]. The main advantage of the former approach is that constructing nodes is inexpensive, since it does not involve any searches. However, this could produce a lot of duplicates and overlaps, making the linearizing operations expensive. Another disadvantage of this approach is that it cannot handle incomplete domains, and can only operate on subtrees. The advantage of the second approach is that the list of nodes is complete and linear at any stage in the algorithm. The drawback, however, is that searching for neighbors is an expensive operation. Our algorithm uses a hybrid approach: it keeps the number of duplicates and overlaps to a minimum and also reduces the search space, thereby reducing the cost of the searching operation. The complete linear octree is first partitioned into coarse blocks using the algorithm described in section 3.1. The descendants of any block, which are present in the fine octree, form a linear subtree with this block as its root. This block-subtree is first balanced using the approach described in section 3.3.2; the size of this tree will be relatively small, and hence the number of duplicates and overlaps will be small too. After balancing all the blocks, the interblock boundaries in each processor are balanced using a variant of the *ripple propagation* algorithm [34] described in section 3.3.4. The performance improvements from using the combined approach are presented in section 4.2.

**3.3.2. Balancing a local block.** In principle, Algorithm 6 can be used to construct a complete balanced subtree of this block for each octant in the initial unbalanced linear subtree. Note that these balanced subtrees may have substantial overlap. Hence, Algorithm 7 is used to remove these overlaps. Lemma 3.1 shows that this process of merging these different balanced subtrees results in a complete linear balanced subtree. However, this implementation would be inefficient due to the number of overlaps, which would in turn increase the storage costs and also make the subsequent operations of sorting and removing duplicates and overlaps more expensive. Instead, we interleave the two operations: constructing the different complete

[13] The number of cells at the boundary depends on the underlying distribution and cannot be known a priori. This further complicates the balancing algorithm.

Algorithm 6. Constructing a complete balanced subtree of an octant, given one of its descendants (sequential).

```
Input:    An octant, $N$, and one of its descendants, $L$.
Output:   Complete balanced subtree, $R$.
Work:     $\mathcal{O}(n \log n)$, where $n =$ len($R$).
Storage:  $\mathcal{O}(n)$, where $n =$ len($R$).

$W \leftarrow L$, $T \leftarrow \emptyset$, $R \leftarrow \emptyset$
for $l \leftarrow \mathcal{L}(L)$ to $(\mathcal{L}(N) + 1)$
    for each $w \in W$
        $R \leftarrow R + w + \mathcal{S}(w)$
        $T \leftarrow T + \{\mathcal{N}(\mathcal{P}(w), l - 1) \cap \{\mathcal{D}(N)\}\}$
    end for
    $W \leftarrow T$, $T \leftarrow \emptyset$
end for
Sort($R$)
RemoveDuplicates($R$)
$R \leftarrow$ Linearise($R$) ( Algorithm 7 )
```

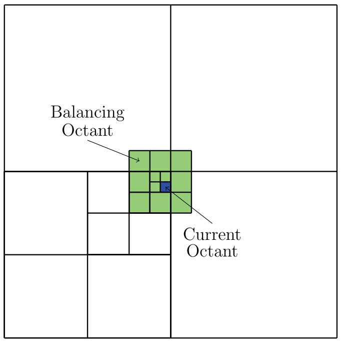


Fig. 6. *The minimal list of balancing quadrants for the current quadrant is shown. This list of quadrants is generated in one iteration of Algorithm* 6.

balanced subtrees and merging them. The overall scheme is described in Algorithm 8.

We note that a list of octants forms a balanced complete octree if and only if for every octant all its neighbors are at the same level as this octant or one level finer or one level coarser. Hence, the coarsest possible octants in a complete octree that will be balanced against this octant are the siblings and the neighbors at the level of this octant's parent. Starting with the finest level and iterating over the levels up to but not including the level of the block, the coarsest possible (without violating the balance constraint) neighbors (Figure 6) of every octant at this level in the current tree (union of the initial unbalanced linear subtree and newly generated octants) are generated. After processing all the octants at any given level, the list of newly introduced coarse octants is merged with the previous list of octants at this level, and duplicate octants are removed. The newly created octants are included while working on subsequent levels. Algorithm 7 still needs to be used in the end to remove overlaps, but the working size is much smaller now compared to the earlier case (Algorithm 6). To avoid redundant work and to reduce the number of duplicates to be removed in the end, we ensure that no two elements in the working list at any given level are siblings of one another. This can be done in a linear pass on the working list for that level, as shown in Algorithm 8.

Algorithm 7. Removing overlaps from a sorted list of octants (sequential) - `Linearize`.

```
Input:     A sorted list of octants, $W$.
Output:    $R$, an octree with no overlaps.
Work:      $\mathcal{O}(n)$, where $n =$ len$(W)$.
Storage:   $\mathcal{O}(n)$, where $n =$ len$(W)$.

for $i \leftarrow 1$ to (len$(W) - 1$)
    if $(W[i] \notin \{\mathcal{A}(W[i+1])\})$
        $R \leftarrow R + W[i]$
    end if
end for
$R \leftarrow R + W[$len$(W)]$
```

Lemma 3.1. *Let $T_1$ and $T_2$ be two complete balanced linear octrees with $n_1$ and $n_2$ number of potential ancestors respectively. Then*

$$T_3 = (T_1 \cup T_2) - \left(\sum_{i=1}^{n_1} \{\mathcal{A}(T_1[i])\}\right) - \left(\sum_{j=1}^{n_2} \{\mathcal{A}(T_2[j])\}\right)$$

*is a complete linear balanced octree.*

*Proof.* $T_4 = (T_1 \cup T_2)$ is a complete octree. Now,

$$\left(\left(\sum_{i=1}^{n_1} \{\mathcal{A}(T_1[i])\}\right) + \left(\sum_{j=1}^{n_2} \{\mathcal{A}(T_2[j])\}\right)\right) = \left(\sum_{k=1}^{n_3} \{\mathcal{A}(T_4[k])\}\right).$$

So,

$$T_3 = \left(T_4 - \left(\sum_{k=1}^{n_3} \{\mathcal{A}(T_4[k])\}\right)\right)$$

is a complete linear octree.

Now, suppose that a node $N \in T_3$ has a neighbor $K \in T_3$ such that $\mathcal{L}(K) \geq (\mathcal{L}(N) + 2)$. It is obvious that exactly one of $N$ and $K$ must be present in $T_1$ and the other must be present in $T_2$. Without loss of generality, assume that $N \in T_1$ and $K \in T_2$. Since $T_2$ is complete, there exists at least one neighbor of $K$,$L \in T_2$, which overlaps $N$. Also, since $T_2$ is balanced $\mathcal{L}(L) = \mathcal{L}(K)$ or $\mathcal{L}(L) = (\mathcal{L}(K) - 1)$ or $\mathcal{L}(L) = (\mathcal{L}(K) + 1)$. So, $\mathcal{L}(L) \geq (\mathcal{L}(N) + 1)$. Since $L$ overlaps $N$ and since $\mathcal{L}(L) \geq (\mathcal{L}(N) + 1)$, $L \in \{\mathcal{D}(N)\}$. Hence, $N \notin T_3$. This contradicts the initial assumption. Therefore, $T_3$ is also balanced. □

**3.3.3. Searching for neighbors.** A leaf needs to be refined if and only if the level of one of its neighbors is at least 2 levels finer than its own. In terms of a search this presents us with two options: search for coarser neighbors or search for finer neighbors. It is much easier to search for coarser neighbors than it is to search for finer neighbors. If we consider the 2-dimensional case, only 3 neighbors coarser than the current cell need to be searched for. However, the number of potential neighbors finer than the cell is extremely large (in two dimensions it is $2 \cdot 2^{D_{max} - l} + 3$, where $l$ is the level of the current quadrant) and therefore not practical to search. In addition the search strategy depends on the way the octree is stored; the pointer-based approach is

Algorithm 8. Balancing a local block (sequential) - `BalanceSubtree`.

**Input:** An octant, $N$, and a partial list of its descendants, $L$.
**Output:** Complete balanced subtree, $R$.
**Work:** $\mathcal{O}(n \log n)$, where $n =$ len$(R)$.
**Storage:** $\mathcal{O}(n)$, where $n =$ len$(R)$.

```
$W \leftarrow L$, $P \leftarrow \emptyset$, $R \leftarrow \emptyset$
for $l \leftarrow D_{max}$ to $(\mathcal{L}(N) + 1)$
    $Q \leftarrow \{x \in W \mid \mathcal{L}(x) = l\}$
    Sort($Q$)
    $T \leftarrow \{x \in Q \mid \mathcal{S}(x) \notin T\}$
    for each $t \in T$
        $R \leftarrow R + t + \mathcal{S}(t)$
        $P \leftarrow P + \{\mathcal{N}(\mathcal{P}(t), l-1) \cap \{\mathcal{D}(N)\}\}$
    end for
    $P \leftarrow P + \{x \in W \mid \mathcal{L}(x) = l - 1\}$
    $W \leftarrow \{x \in W | \mathcal{L}(x) \neq l - 1\}$
    RemoveDuplicates($P$)
    $W \leftarrow W + P$, $P \leftarrow \emptyset$
end for
Sort($R$)
$R \leftarrow$ Linearise($R$) ( Algorithm 7 )
```

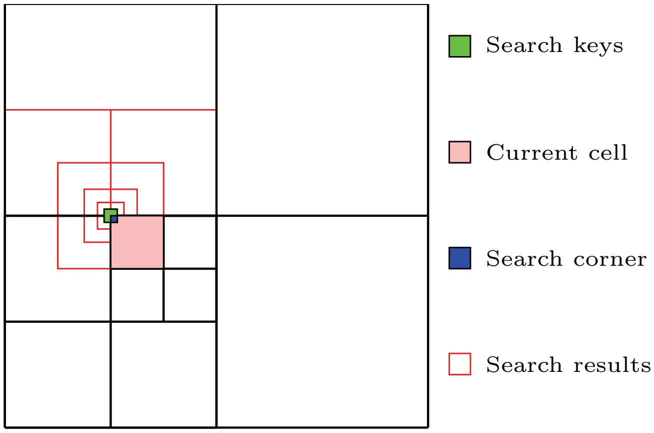


Fig. 7. *To find neighbors coarser than the current cell, we first select the finest cell at the far corner. The far corner is the one that is not shared with any of the current cell's siblings. The neighbors of this corner cell are determined and used as the search keys. The search returns the greatest cell lesser than or equal to the search key. The possible candidates in a complete linear quadtree, as shown, are ancestors of the search key.*

more popular [4, 32], but has the overhead that it has to be rebuilt every time octants are communicated across processors. In the proposed approach the octree is stored as a linear octree in which the octants are sorted globally in the ascending Morton order, allowing us to search in $\mathcal{O}(\log n)$.

In order to find neighbors coarser than the current cell, we use the approach illustrated in Figure 7. First, the finest cell at the far corner (marked as "search corner" in Figure 7) is determined. This is the corner that this octant shares with its parent. This is also the corner diagonally opposite to the corner common to all the siblings of the current cell.[14] The neighbors (at the finest level) of this cell ($N$) are then selected and used as the search keys. These are denoted by $\mathcal{N}^s\left(N, D_{max}\right)$. The maximum lower bound[15] for the given search key is determined by searching within the complete linear octree. In a complete linear octree, the maximum lower bound of a search key returns its finest ancestor. If the search result is at a level finer than or equal to the current cell, then it is guaranteed that no coarser neighbor can exist in that direction. This idea can be extended to incomplete linear octrees (including multiply connected domains). In this case, the result of a search is ignored if it is not an ancestor of the search key.

**3.3.4. Ripple propagation.** A variant (Algorithm 9) of the *prioritized ripple propagation* algorithm first proposed by Tu and O'Hallaron [32], modified to work with linear octrees, is used to balance the boundary leaves. The algorithm selects all leaves at a given level (successively decreasing levels starting with the finest) and searches for neighbors coarser than itself. A list of balancing descendants[16] for neighbors that violate the balance condition is stored. At the end of each level, any octant that violated the balance condition is replaced by a complete linear subtree. This subtree can be obtained either by using the sequential version of Algorithm 4 or by using Algorithm 10, which is a variant of Algorithm 8. Both algorithms perform equally well.[17]

One difference with earlier versions of the ripple propagation algorithm is that our version works with incomplete domains. In addition, earlier approaches [4, 32, 34] have used pointer-based representations of the local octree, which incurs the additional cost of constructing the pointer-based tree from the linear representation and also increases the memory footprint of the octree as 9 additional pointers[18] are required per octant. The work and storage costs incurred for balancing using the proposed algorithm to construct $n$ balanced octants are $\mathcal{O}(n \log n)$ and $\mathcal{O}(n)$, respectively. This is true irrespective of the domain, including domains that are not simply connected.

**3.3.5. Insulation against the ripple effect.** An interesting property of complete linear octrees is that a boundary octant cannot be finer than its internal neighbors[19] (Figure 8(a)) [32]. So, if a node (at any level) is internally balanced, then to balance it with all its neighboring domains, it is sufficient to appropriately refine the internal boundary leaves.[20] The interior leaves need not be refined any further. Since the interior leaves are also balanced against all their neighbors, they will not force any other octant to split. Hence, interior octants do not participate in the remaining stages of balancing.

Observe that the phenomenon with interior octants described above is only an example of a more general property.

[14] We do not need to search in the direction of the siblings.

[15] The greatest cell lesser than or equal to the search key is referred to as its maximum lower bound.

[16] Balancing descendants are the minimum number of descendants that will balance against the octant that performed the search.

[17] We indicate which algorithms are parallel and which are sequential. In our notation the sequential algorithms are sometimes invoked with a distributed object: it is implied that the input is the local instance of the distributed object.

[18] One pointer to the parent and eight pointers to its children.

[19] A neighbor of a boundary octant that does not touch the boundary is referred to as an internal neighbor of the boundary octant.

[20] We refer to the descendants of a node that touch its boundary from the inside as its internal boundary leaves.

Algorithm 9. Ripple propagation on incomplete domains (sequential) - `Ripple`.

```
Input:     $L$, a sorted incomplete linear octree.
Output:    $W$, a balanced incomplete linear octree.
Work:      $\mathcal{O}(n \log n)$, where $n =$ len($L$).
Storage:   $\mathcal{O}(n)$, where $n =$ len($L$).

$W \leftarrow L$
for $l \leftarrow D_{max}$ to $3$
    $T, R \leftarrow \emptyset$
    for each $w \in W$
        if $\mathcal{L}(w) = l$
            $K \leftarrow$ search_keys($w$) ( Section 3.3.3 )
            $(B, J) \leftarrow$ maximum_lower_bound ($K$, $W$)
                    ($J$ is the index of $B$ in $W$)
            for each $(b, j) \in (B, J) \mid (\exists\, k \in K \mid b \in \{\mathcal{A}(k)\})$
                $T[j] \leftarrow T[j] + (\{\mathcal{N}^s(w, (l-1))\} \cap \{\mathcal{D}(b)\})$
            end for
        end if
    end for
    for $i \leftarrow 1$ to len($W$)
        if $T[i] \neq \emptyset$
            $R \leftarrow R +$ CompleteSubtree($W[i]$, $T[i]$) ( Algorithm 10 )
        else
            $R \leftarrow R + W[i]$
        end if
    end for
    $W \leftarrow R$
end for
```

Algorithm 10. Completing a local block (sequential) - `CompleteSubtree`.

```
Input:     An octant, $N$, and a partial list of its descendants, $L$.
Output:    Complete subtree, $R$.
Work:      $\mathcal{O}(n \log n)$, where $n =$ len($R$).
Storage:   $\mathcal{O}(n)$, where $n =$ len($R$).

$W \leftarrow L$
for $l \leftarrow D_{max}$ to $\mathcal{L}(N) + 1$
    $Q \leftarrow \{x \in W \mid \mathcal{L}(x) = l\}$
    Sort($Q$)
    $T \leftarrow \{x \in Q \mid \mathcal{S}(x) \notin T\}$
    for each $t \in T$
        $R \leftarrow R + t + \mathcal{S}(t)$
        $P \leftarrow P + \mathcal{S}(\mathcal{P}(t))$
    end for
    $P \leftarrow P + \{x \in W \mid \mathcal{L}(x) = l - 1\}$
    $W \leftarrow \{x \in W \mid \mathcal{L}(x) \neq l - 1\}$
    RemoveDuplicates($P$)
    $W \leftarrow W + P$, $P \leftarrow \emptyset$
end for
Sort($R$)
$R \leftarrow$ Linearise($R$) ( Algorithm 7 )
```

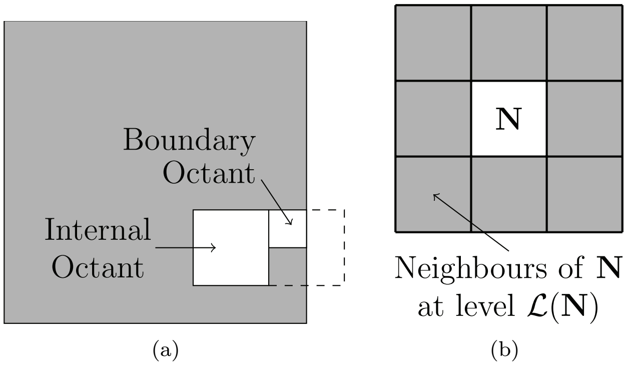


FIG. 8. (a) *A boundary octant cannot be finer than its internal neighbors, and* (b) *an illustration of an insulation layer around octant* $\boldsymbol{N}$. *No octant outside this layer of insulation can force a split on* $\boldsymbol{N}$.

DEFINITION 2. *For any octant, $N$, in the octree, we refer to the union of the domains occupied by its potential neighbors at the same level as $N$ ($\mathcal{N}(N, \mathcal{L}(N))$) as the insulation layer around octant $N$. This will be denoted by $\mathcal{I}(N)$.*

PROPERTY 1. *No octant outside the insulation layer around octant $N$ can force $N$ to split (Figure* 8(b)*).*

This property allows us to decouple the problem of balancing and allows us to work on only a subset of nodes in the octree and yet ensure that the entire octree is balanced.

**3.3.6. Balancing interprocessor boundaries.** After the intraprocessor and interblock boundaries are balanced, the interprocessor boundaries need to be balanced. Unlike the internal leaves (section 3.3.5), the octants on the boundary do not have any insulation against the ripple effect. Moreover, a ripple can propagate across multiple processors. Most approaches to performing this balance have been based on extensions of the sequential ripple algorithm to a parallel case by performing parallel searches. In an earlier attempt we developed efficient parallel search strategies allowing us to extend our sequential balancing algorithms to the parallel case. Although this approach works well for small problems on a small number of processors, it shows suboptimal isogranular scalability, as has been seen with other similar approaches to the problem [34]. The main reason is iterative communication. Although there are many examples of scalable parallel algorithms that involve iterative communication, they overlap communication with computation to reduce the overhead associated with communication [12, 28]. Currently, there is no method that overlaps communication with computation for the balancing problem. Thus, any algorithm that uses iterative parallel searches for balancing octrees will have high communication costs.

In order to avoid parallel searches, the problem of balancing is decoupled. In other words, each processor works independently without iterative communication. To achieve this, two properties are used: (1) The only octants that need to be refined after the local balancing stage are the ones whose insulation layer is not contained entirely within the same processor; we will refer to them as the unstable octants. (2) An artificial insulation layer (Property 1) for these octants can be constructed with little communication overhead (section 3.3.7).

Note that although it is sufficient to build an insulation layer for octants that truly touch the interprocessor boundary, it is nontrivial to identify such octants. Moreover,

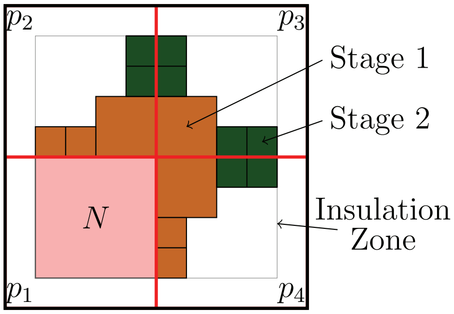


FIG. 9. *Communication for interprocessor balancing is done in two stages: First, every octant on the interprocessor boundary (stage* 1*) is communicated to processors that overlap with its insulation layer. Next, all the local interprocessor boundary octants that lie in the insulation layer of a remote octant (*$N$*) received from another processor are communicated to that processor (stage* 2*).*

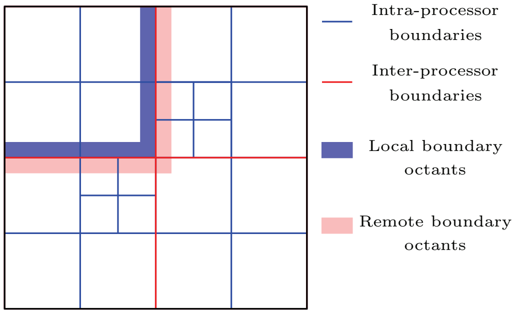


FIG. 10. *A coarse quadtree illustrating inter- and intraprocessor boundaries. First, every processor balances each of its local blocks. Then each processor balances the cells on its intraprocessor boundaries. The octants that lie on interprocessor boundaries are then communicated to their respective processors and each processor balances the combined list of local and remote octants.*

even if it was easy to identify the true interprocessor boundary octants, all unstable octants must participate in subsequent balancing as well. Hence, the insulation layer is built for all unstable octants as they can be identified easily. Since most of the unstable octants do touch the interprocessor boundaries, we will simply refer to them as interprocessor boundary octants in the following sections.

The construction of the insulation layer for the interprocessor boundary octants is done in two stages (Figure 9): First, every local octant on the interprocessor boundary (Figure 10) 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 interprocessor 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. For simplicity, Algorithm 11 only describes a naïve implementation for determining the octants that need to be communicated in this stage. However, this can be performed much more efficiently using the results of Lemmas 3.2 and 3.3. After this two-stage communication, each processor balances the union of the local and remote boundary octants using the ripple propagation–based method (section 3.3.4). At the end only the octants spanning the original domain spanned by the processors are re-

Algorithm 11. Balancing complete linear octrees (parallel).

```
Input:      A distributed sorted complete linear octree, $L$.
Output:     A distributed complete balanced linear octree, $R$.
Work:       $\mathcal{O}(n \log n)$, where $n =$ len($L$).
Storage:    $\mathcal{O}(n)$, where $n =$ len($L$).
Time:       Refer to section 3.3.7.

1.   $B \leftarrow$ BlockPartition($L$) ( Algorithm 2 )
2.   $C \leftarrow \emptyset$
3.   for each $b \in B$
4.       $C \leftarrow C+$ BalanceSubtree($b$, $\{\{\mathcal{D}(b)\} \cap L\}$) ( Algorithm 8 )
5.   end for
6.   $D \leftarrow \{x \in C \mid \exists\ z \in \{\mathcal{I}(x)\} \mid \mathcal{B}(z) \neq \mathcal{B}(x)\}$
         ( intraprocessor boundary octants )
7.   $S \leftarrow$ Ripple($D$) ( Algorithm 9 )
8.   $F \leftarrow (C - D) \cup S$
9.   $G \leftarrow \{x \in S \mid \exists\ z \in \{\mathcal{I}(x)\} \mid$ rank($z$) $\neq$ rank($x$) $\}$
         ( interprocessor boundary octants )
10.  for each $g \in G$
11.      for each $b \in B_{global} - B$
12.          if $\{b \cap \mathcal{I}(g)\} \neq \emptyset$
13.              Send($g$, rank($b$))
14.          end if
15.      end for
16.  end for
17.  $T \leftarrow$ Receive()
18.  for each $g \in G$
19.      for each $t \in T$
20.          if $\{g \cap \mathcal{I}(t)\} \neq \emptyset$
21.              if $g$ was not sent to rank($t$) in Step 10
22.                  Send($g$, rank($t$))
23.              end if
24.          end if
25.      end for
26.  end for
27.  $K \leftarrow$ Receive()
28.  $H \leftarrow$ Ripple($G \cup T \cup K$)
29.  $R \leftarrow \{x \in \{H \cup F\} \mid \{B \cap \{x, \{\mathcal{A}(x)\}\}\} \neq \emptyset\}$
30.  $R \leftarrow$ Linearise($R$) ( Algorithm 7 )
```

tained. Although there is some redundancy in the work, it is compensated for by the fact that we avoid iterative communications. Section 3.3.7 gives a detailed analysis of the communication cost involved.

Lemma 3.2. *If octants $a$ and $b > a$ do not overlap, then there can be no octant $c > b$ that overlaps $a$.*

*Proof.* If $a$ and $c$ overlap, then either $a \in \{\mathcal{A}(c)\}$ or $a \in \{\mathcal{D}(c)\}$. Since $c > a$, the latter is a direct violation of Property 4 and hence is impossible. Hence, assume that $c \in \{\mathcal{D}(a)\}$. By Property 9, $c \leq \mathcal{DLD}(a)$. Property 10 would then imply that $b \in \{\mathcal{D}(a)\}$. Property 5 would then imply that $a$ and $b$ must overlap. Since this is not true our initial assumption must be wrong. Hence, $a$ and $c$ cannot overlap. □

Lemma 3.3. *Let $N$ be an interprocessor boundary octant belonging to processor $q$. If the $\mathcal{I}(N)$ is contained entirely within processors $q$ and $p$, then the interprocessor boundary octants on processor $p$ that overlap with $\mathcal{I}(N)$ and that were not communicated to $q$ in the first stage will not force a split on $N$.*

*Proof.* Note that at this stage both $p$ and $q$ are internally balanced. Thus, $N$ will be forced to split if and only if there is a true interprocessor boundary octant,

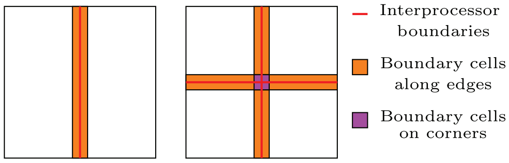

Interprocessor boundaries
Boundary cells along edges
Boundary cells on corners

Fig. 11. *Cells that lie on the interprocessor boundaries. The figure on the left shows an interprocessor boundary involving* 2 *processors and the figure on the right shows an interprocessor boundary involving* 4 *processors.*

$a$, on $p$ touching an octant, $b$, on $q$ such that $\mathcal{L}(a) > (\mathcal{L}(b) + 1)$, and when $b$ is split it starts a cascade of splits on octants in $q$ that in turn force $N$ to split. Since every true interprocessor boundary octant is sent to all its adjacent processors, $a$ must have been sent to $q$ during the first stage of communication. □

Algorithm 11 gives the pseudocode for the overall parallel balancing.

**3.3.7. Communication costs for parallel balancing.** Although not all unstable octants are true interprocessor boundaries, it is easier to visualize and understand the arguments presented in this section if this subtle point is ignored. Moreover, since we only compare the communication costs associated with the two approaches (up-front communication versus iterative communication) and since the majority of unstable octants are true interprocessor boundary octants, it is not too restrictive to assume that all unstable octants are true interprocessor boundary octants.

Let us assume that prior to parallel balancing there are a total of $N$ octants in the global octree. The octants that lie on the interprocessor boundary can be classified based on the *degree of the face*[21] that they share with the interprocessor boundary. We use $N_k$ to represent the number of octants that touch any $m$-dimensional face ($m \in [0, k]$) of the interprocessor boundary.

Note that all vertex boundary octants are also edge and face boundaries and that all edge boundary octants are also face boundary octants. Therefore we have $N \geq N_2 \geq N_1 \geq N_0$, and for $N \gg n_p$, we have $N \gg N_2 \gg N_1 \gg N_0$.

Although it is theoretically possible that an octant is larger than the entire domain controlled by some processors, it is unlikely for dense octrees. Thus, ignoring such cases we can show that the total number of octants of a $d$-tree that need to be communicated in the first stage of the proposed approach is given by

$$N_u = \sum_{k=1}^{d} 2^{d-k} N_{k-1}. \tag{3.1}$$

Consider the example shown in Figure 11. The domain on the left is partitioned into two regions, and in this case all boundary octants need to be transmitted to exactly one other processor. The addition of the additional boundary, in the figure on the right, does not affect most boundary nodes, except for the boundary octants that share a corner, i.e., a 0-dimensional face with the interprocessor boundaries. These octants need to be sent to an additional 2 processors, and that is the reason we have a factor of $2^{d-k}$ in (3.1). For the case of octrees, additional communication is incurred because of edge boundaries as well as vertex boundaries. Edge boundary octants need to be communicated to 2 additional processors, whereas the vertex boundary octants need to be communicated to 4 additional processors (7 processors in all).

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

Now, we analyze the cost associated with the second communication step in our algorithm. Consider the example shown in Figure 9. Note that all the immediate neighbors of the octant under consideration (octant on processor 1 in the figure) were communicated during the first stage. The octants that lie in the insulation zone of this octant and that were not communicated in the first stage are those that lie in a direction normal to the interprocessor boundary. However, most octants that lie in a direction normal to the interprocessor boundary are internal octants on other processors. As shown in Figure 9, the only octants that lie in a direction normal to one interprocessor boundary and are also tangential to another interprocessor boundary are the ones that lie in the shadow of some edge or corner boundary octant. Therefore, we communicate only $\mathcal{O}(N_1 + N_0)$ octants during this stage. Since $N \gg n_p$ and $N_2 \gg N_1 \gg N_0$ for most practical applications, the cost for this communication step can be ignored.

The minimum number of search keys that need to be communicated in a search-based approach is given by

$$N_s = \sum_{k=1}^{d} 2^{k-1} N_{k-1}. \tag{3.2}$$

Again considering the example shown in Figure 11 each boundary octant in the figure shown on the left generates 3 search keys, out of which one lies on the same processor. The other two need to be communicated to the other processor. The addition of the extra boundary, in the figure on the right, does not affect most boundary nodes, except for the boundary octants that share a corner, i.e., a 0-dimensional face with the interprocessor boundaries. These octants need to be sent to an additional processor, and that is the reason we have a factor of $2^{k-1}$ in (3.2). It is important to observe the difference between the communication estimates for upfront communication, (3.1), with that of the search based approach, (3.2). For large octrees,

$$N_u \approx N_2,$$

while

$$N_s \approx 4N_2.$$

Note that in arriving at the communication estimate for the search-based approaches, we have not accounted for the additional octants created during the interprocessor balancing. In addition, iterative search-based approaches are further affected by communication lag and synchronization. Our approach in contrast requires no subsequent communication.

In conclusion, the communication cost involved in the proposed approach is lower than that of search-based approaches.[22]

## 4. Results.

The performance of the proposed algorithms is evaluated by a number of numerical experiments, including fixed-size and isogranular scalability analysis. The algorithms were implemented in C++ using the MPI library. A variant of the sample sort algorithm was used to sort the points and the octants, which incorporates a parallel bitonic sort to sort the sample elements as suggested in [12]. PETSc [2] was used for profiling the code. All tests were performed on the Pittsburgh Supercomputing Center's TCS-1 terascale computing HP AlphaServer Cluster comprising 750

[22] We are assuming that both approaches use the same partitioning of octants.

TABLE 3

*Input and output sizes for the construction and balancing algorithms for the scalability experiments on Gaussian, log-normal, and regular point distributions. The output of the construction algorithm is the input for the balancing algorithm. All the octrees were generated using the same parameters: $D_{max} = 30$ and $N^p_{max} = 1$. Differences in the number and distributions of the input points result in different octrees for each case. The maximum level of the leaves for each case is listed. Note that none of the leaves produced was at the maximum permissible depth ($D_{max}$). This depends only on the input distribution. Regular point distributions are inherently balanced, and so we report the number of octants only once.*

| Problem size | Gaussian | | | | Log-normal | | | | Regular | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Points | Balancing | | Max. Level ($\mathcal{L}^*$) | Points | Balancing | | $\mathcal{L}^*$ | Points | Leaves | $\mathcal{L}^*$ |
| | | Leaves before | Leaves after | | | Leaves before | Leaves after | | | | |
| 1M | 180K | 607K | 0.99M | 14 | 180K | 607K | 0.99M | 13 | 0.41M | 0.99M | 7 |
| 2M | 361K | 1.2M | 2M | 15 | 361K | 1.2M | 2M | 14 | 2M | 2M | 7 |
| 4M | 720K | 2.4M | 3.9M | 14 | 720K | 2.4M | 3.9M | 15 | 2.4M | 4.06M | 8 |
| 8M | 1.5M | 4.9M | 8.0M | 16 | 1.5M | 4.9M | 8.1M | 16 | 3.24M | 7.96M | 8 |
| 16M | 2.9M | 9.7M | 16M | 16 | 2.9M | 9.7M | 16M | 16 | 16.8M | 16.8M | 8 |
| 32M | 5.8M | 19.6M | 31.9M | 17 | 5.8M | 19.6M | 31.8M | 17 | 19.3M | 32.5M | 9 |
| 64M | 11.7M | 39.3M | 64.4M | 18 | 11.7M | 39.3M | 64.7M | 17 | 25.9M | 63.7M | 9 |
| 128M | 23.5M | 79.3M | 0.13B | 19 | 23.5M | 79.4M | 0.13B | 19 | 0.13B | 0.13B | 9 |
| 256M | 47M | 0.16B | 0.26B | 19 | 47M | 0.16B | 0.26B | 19 | 0.15B | 0.26B | 10 |
| 512M | 94M | 0.32B | 0.52B | 20 | 94M | 0.32B | 0.52B | 20 | 0.17B | 0.34B | 10 |
| 1B | 0.16B | 0.55B | 0.91B | 21 | 0.16B | 0.55B | 0.91B | 20 | 1.07B | 1.07B | 10 |

SMP ES45 nodes. Each node is equipped with four Alpha EV-68 processors at 1 GHz and 4 GB of memory. The peak performance is approximately 6 Tflops, and the peak performance for the top-500 LINPACK benchmark is approximately 4 Tflops. The nodes are connected by a Quadrics interconnect, which delivers over 500 MB/s of message-passing bandwidth per node and has a bisection bandwidth of 187 GB/s. In our tests, we have used 4 processors per node wherever possible.

We present results from an experiment that we conducted to highlight the advantage of using the proposed two-stage method for intraprocessor balancing. Also, we present fixed-size and isogranular scalability analysis results.

**4.1. Test data.** Data of different sizes were generated for three different spatial distributions of points: Gaussian, log-normal, and regular. The regular distribution corresponds to a set of points distributed on a Cartesian grid. Datasets of increasing sizes were generated for all three distributions so that they result in balanced octrees with octants ranging from $10^6$(1M) to $10^9$(1B). All of the experiments were carried out using the same parameters: $D_{max} = 30$ and $N^p_{max} = 1$. Only the number and distribution of points were varied to produce the various octrees. The fixed-size scalability analysis was performed by selecting the 1M, 32M, and 128M Gaussian point distributions to represent small, medium, and large problems. We provide the input and output sizes for the construction and balancing algorithms in Table 3. The output of the construction algorithm is the input for the balancing algorithm.

**4.2. Comparison between different strategies for the local balancing stage.** In order to assess the advantages of using a two-stage approach for local balancing over existing methods, we compared the runtimes on different problem sizes. Since the comparison was for different local-balancing strategies, it does not involve any communication and hence was evaluated on a shared memory machine. We compared our two-stage approach, discussed in section 3.3.1, with two other approaches; the first approach is the prioritized ripple propagation idea applied on the entire local domain [34], and the second approach is to use ripple propagation in 2 stages, where

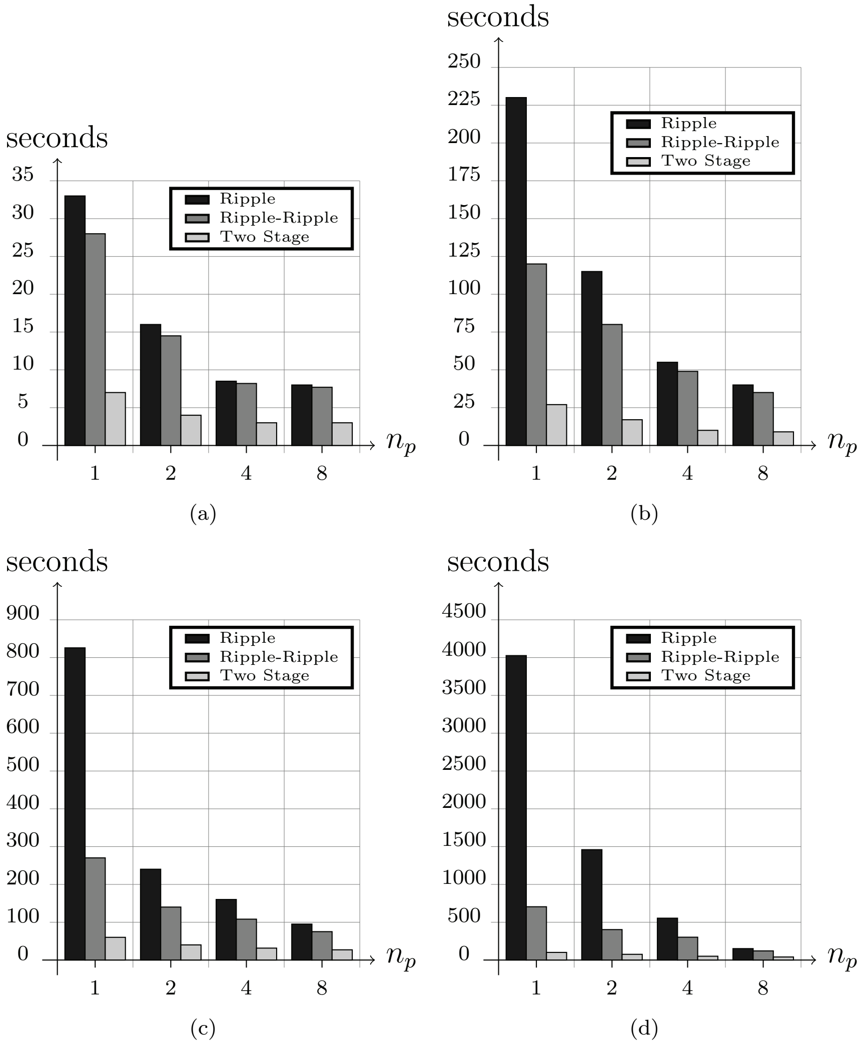
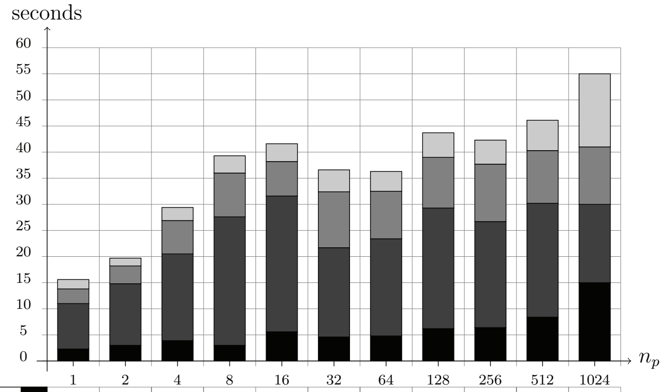


FIG. 12. *Comparison of three different approaches for balancing linear octrees* (a) *for a Gaussian distribution of* 1M *octants,* (b) *for a Gaussian distribution of* 4M *octants,* (c) *for a Gaussian distribution of* 8M *octants, and* (d) *for a Gaussian distribution of* 16M *octants.*

the local domain is first split into coarser blocks[23] and ripple propagation is applied first to each local block and then repeated on the boundaries of all local blocks. Fixed-size scalability analysis was performed to compare the above-mentioned three approaches with problem sizes of 1, 4, 8, and 16 million octants. The results are shown in Figure 12. All three approaches demonstrate good fixed-size scalability, but the proposed two-stage approach has a lower absolute runtime.

**4.3. Scalability analysis.** In this section, we provide experimental evidence of the good scalability of our algorithms. We present both fixed-size and isogranular scalability analysis. Fixed-size scalability was performed for different problem sizes

[23] The same partitioning strategy as used in our two-stage algorithm was used to obtain the coarser blocks.



| | | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Construction | ■ | 2.3 | 3.0 | 3.9 | 3.0 | 5.6 | 4.6 | 4.8 | 6.2 | 6.4 | 8.4 | 15 |
| Internal Balance | ■ | 8.7 | 11.8 | 16.6 | 24.6 | 26.0 | 17.1 | 18.6 | 23.1 | 20.3 | 21.8 | 15 |
| Boundary Balance | ▨ | 2.8 | 3.4 | 6.4 | 8.4 | 6.6 | 10.7 | 9.1 | 9.7 | 11.0 | 10.1 | 11 |
| Communication for Balance | □ | 1.8 | 1.5 | 2.5 | 3.3 | 3.4 | 4.2 | 3.8 | 4.7 | 4.6 | 5.8 | 14 |

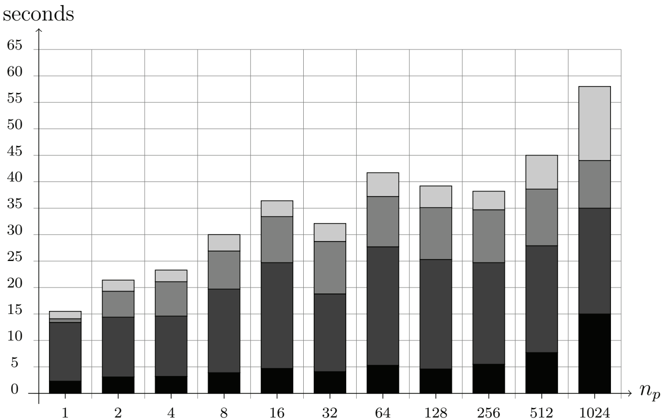
FIG. 13. *Isogranular scalability for a Gaussian distribution of* 1M *octants per processor. From left to right, the bars indicate the time taken for the different components of our algorithms for increasing processor counts. The bar for each processor is partitioned into* 4 *sections. From top to bottom, the sections represent the time taken for* (1) *communication (including related preprocessing and postprocessing) during balance refinement (Algorithm* 11*),* (2) *balancing across intra and inter processor boundaries (Algorithm* 9*),* (3) *balancing the blocks (Algorithm* 8*), and* (4) *construction from points (Algorithm* 5*).*

to compute the speedup when the problem size is kept constant and the number of processors is increased. Isogranular scalability analysis is performed by tracking the execution time while increasing the problem size and the number of processors proportionately. By maintaining the problem size per processor (relatively) constant as the number of processors is increased, we can identify communication problems related to the size and frequency of the messages as well as global reductions and problems with algorithmic scalability.

One of the important components in our algorithms is the sample sort routine, which has a complexity of $\mathcal{O}(\frac{N}{n_p}\log\frac{N}{n_p} + n_p^2\log n_p)$ if the samples are sorted using a serial sort. This causes problems when $\mathcal{O}(N) < \mathcal{O}(n_p^3)$ as the serial sort begins to dominate and results in poor scalability. For example, at $n_p = 1024$ we would require $\frac{N}{n_p} > 10^6$ to obtain good scalability. This presents some problems as it becomes difficult to fit arbitrarily large problems on a single processor. A solution, first proposed in [12], is to sort the samples using the parallel bitonic sort. This approach reduces the complexity of sorting to $\mathcal{O}(\frac{N}{n_p}\log\frac{N}{n_p} + n_p\log n_p)$.

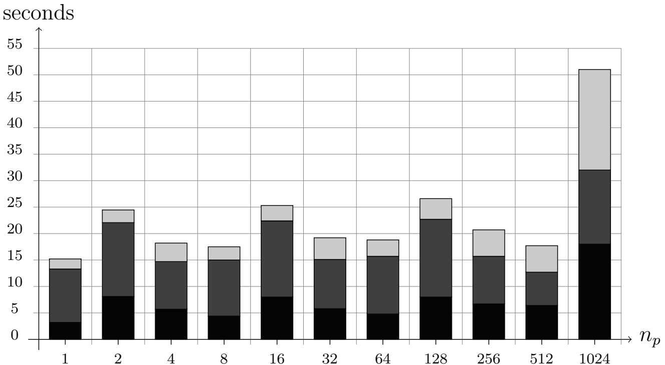
Isogranular scalability analysis was performed for all three distributions with an output size of roughly 1M octants per processor, for processor counts ranging from 1 to 1024. Wall-clock timings, speedup, and efficiency for the isogranular analysis for the three distributions are shown in Figures 13, 14, and 15.

Since the regularly spaced distribution is inherently balanced, the input point sizes were much greater for this case than those for Gaussian and log-normal distributions. Both the Gaussian and log-normal distributions are imbalanced; and in Table 3, we can see that, on average, the number of unbalanced octants is three times the number of input points and the number of octants doubles after balancing. For the regularly spaced distribution, we observe that in some cases the number of octants is the same as the number of input points (2M, 16M, 128M, and 1B). These are special cases where



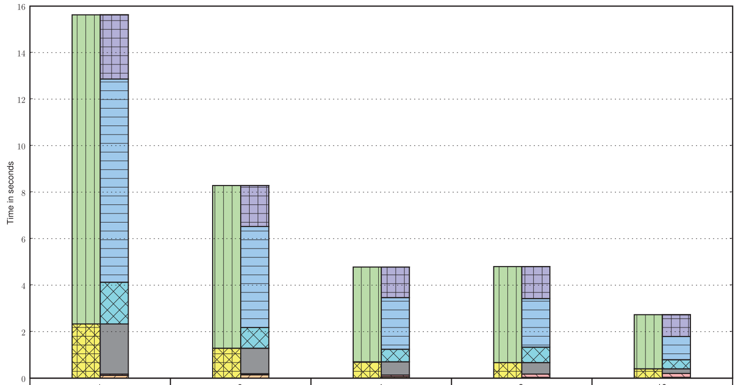
| | | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Construction | ■ | 2.3 | 3.1 | 3.2 | 3.9 | 4.7 | 4.1 | 5.3 | 4.6 | 5.5 | 7.7 | 15 |
| Internal Balance | ▩ | 11.1 | 11.3 | 11.4 | 15.8 | 20.0 | 14.7 | 22.4 | 20.7 | 19.2 | 20.2 | 20 |
| Boundary Balance | ▨ | 0.7 | 4.9 | 6.5 | 7.2 | 8.7 | 9.9 | 9.5 | 9.8 | 10.0 | 10.7 | 9 |
| Communication for Balance | □ | 1.4 | 2.1 | 2.2 | 3.1 | 3.0 | 3.4 | 4.5 | 4.1 | 3.5 | 6.4 | 14 |

FIG. 14. *Isogranular scalability for a log-normal 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) *communication (including related preprocessing and postprocessing) during balance refinement (Algorithm* 11*),* (2) *balancing across intra- and interprocessor boundaries (Algorithm* 9*),* (3) *balancing the blocks (Algorithm* 8*), and* (4) *construction from points (Algorithm* 5*).*



| | | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Construction | ■ | 3.2 | 8.1 | 5.7 | 4.4 | 8.0 | 5.8 | 4.8 | 8.0 | 6.7 | 6.4 | 18 |
| Internal Balance | ▩ | 10.1 | 13.96 | 9.0 | 10.6 | 14.4 | 9.3 | 10.9 | 14.7 | 9.0 | 6.3 | 14 |
| Boundary Balance | ▨ | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| Communication for Balance | □ | 1.9 | 2.4 | 3.5 | 2.5 | 2.9 | 4.1 | 3.1 | 3.9 | 5.0 | 5.0 | 19 |

FIG. 15. *Isogranular scalability for a regular 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) *communication (including related preprocessing and postprocessing) during balance refinement (Algorithm* 11*),* (2) *balancing across intra- and interprocessor boundaries (Algorithm* 9*),* (3) *balancing the blocks (Algorithm* 8*), and* (4) *construction from points (Algorithm* 5*). While both the input and output grain sizes remain almost constant for the Gaussian and log-normal distributions, only the output grain size remains constant for the uniform distribution. Hence, the trend seen in this study is a little different from those for the Gaussian and log-normal distributions.*



| | | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|---|
| Construction | 🟨 | 2.3 | 1.3 | 0.7 | 0.7 | 0.4 |
| Sort - Construction | 🟧 | 0.1 | 0.1 | 0.1 | 0.0 | 0.0 |
| BlockPartition | 🟥 | 0.1 | 0.0 | 0.1 | 0.1 | 0.2 |
| Balance | 🟩 | 13.3 | 7.0 | 4.1 | 4.1 | 2.3 |
| Commn - Balance | 🟦 | 1.8 | 0.9 | 0.5 | 0.7 | 0.4 |
| Internal Balance | 🟦 | 8.7 | 4.3 | 2.2 | 2.1 | 1.0 |
| Boundary Balance | 🟪 | 2.8 | 1.8 | 1.3 | 1.4 | 0.9 |

FIG. 16. *Fixed-size scalability for a Gaussian distribution of* 1M *octants. 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* 2 *columns, which are further subdivided. The left column is subdivided into* 2 *sections and the right column is subdivided into* 6 *sections. The top and bottom sections of the left column represent the total time taken for* (1) *balance refinement (Algorithm* 11*) and* (2) *construction (Algorithm* 5*), respectively. From top to bottom, the sections of the right column represent the time taken for* (1) *balancing across intra- and interprocessor boundaries (Algorithm* 9*),* (2) *balancing the blocks (Algorithm* 8*),* (3) *communication (including related preprocessing and postprocessing) during balance refinement,* (4) *local processing during construction,* (5) `BlockPartition`*, and* (6) `Sample Sort`*.*

the resulting grid is a perfect regular grid. Thus, while both the input and output grain sizes remain almost constant for the Gaussian and log-normal distributions, only the output grain size remains constant for the regular distribution. Hence, the trend for the regular distribution is a little different from those for the Gaussian and log-normal distributions.

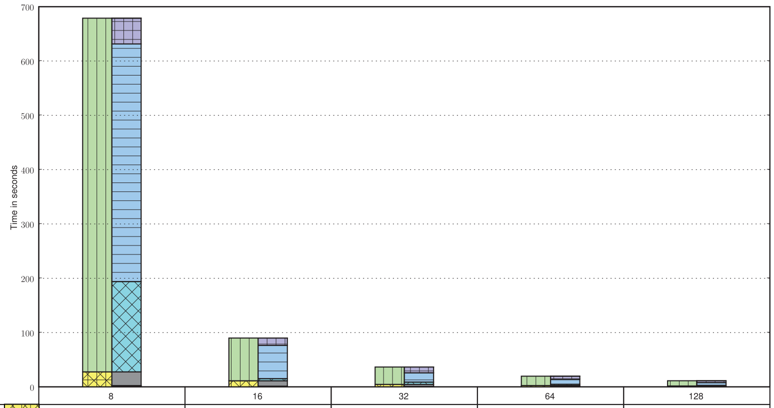
The plots demonstrate the good isogranular scalability of the algorithm. We achieve near optimal isogranular scalability for all three distributions (50s per $10^6$ octants per processor for the Gaussian and log-normal distributions and 25s for the regularly spaced distribution).

Fixed-size scalability tests were also performed for three problem set sizes, small (1 million points), medium (32 million points), and large (128 million points) for the Gaussian distribution. These results are plotted in Figures 16, 17, and 18.

## 5. Conclusions.

We have presented two new parallel algorithms for constructing and balancing large linear octrees on distributed memory machines. We have also tested MPI-based scalable parallel implementations for both the algorithms. Our algorithms have several important features:

- Experiments on three different types of input distributions demonstrate that the algorithms are insensitive to the underlying data distribution.
- Our algorithms avoid iterative communications and thus are able to achieve low absolute runtime and good scalability.
- Experiments demonstrate that the proposed two-stage intraprocessor balancing algorithm has a significantly lower running time compared to alternative approaches.



| | | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| Construction | ▨ | 27.7 | 11.3 | 4.6 | 2.5 | 1.7 |
| Sort - Construction | ▨ | 1.7 | 0.8 | 0.4 | 0.2 | 0.2 |
| BlockPartition | ▨ | 0.8 | 0.5 | 0.4 | 0.4 | 0.6 |
| Balance | ▨ | 651.2 | 78.7 | 32.0 | 17.2 | 9.6 |
| Commn - Balance | ▨ | 166.2 | 3.9 | 4.2 | 2.3 | 1.1 |
| Internal Balance | ▨ | 437.5 | 61.2 | 17.1 | 9.5 | 5.0 |
| Boundary Balance | ▨ | 47.4 | 13.6 | 10.7 | 5.3 | 3.4 |

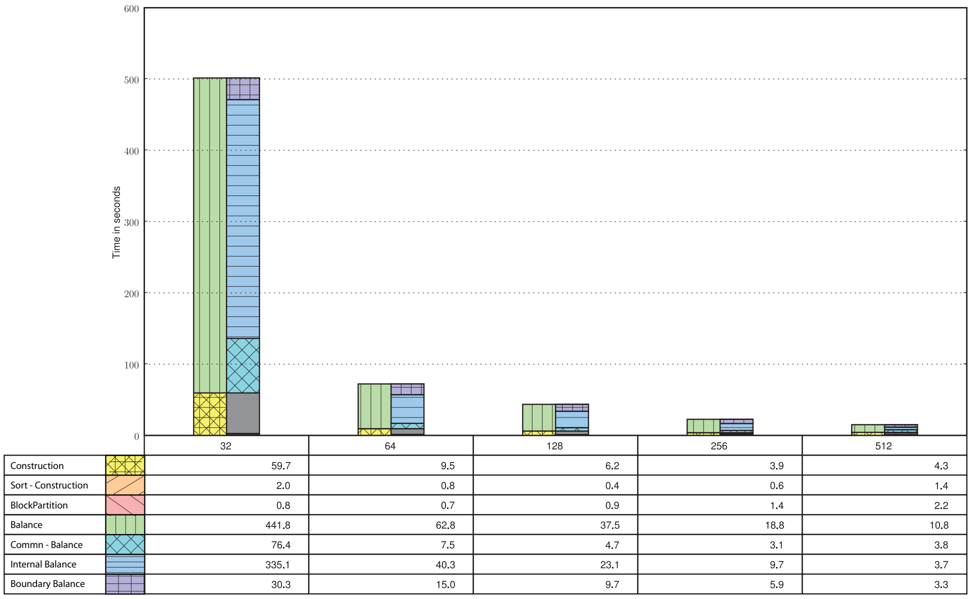
FIG. 17. *Fixed-size scalability for a Gaussian distribution of* 32M *octants. From left to right, the bars indicate the time taken for the different components of our algorithms for increasing processor counts. The bar for each processor is partitioned into* 2 *columns, which are further subdivided. The left column is subdivided into* 2 *sections and the right column is subdivided into* 6 *sections. The top and bottom sections of the left column represent the total time taken for* (1) *balance refinement (Algorithm* 11*) and* (2) *construction (Algorithm* 5*), respectively. From top to bottom, the sections of the right column represent the time taken for* (1) *balancing across intra- and interprocessor boundaries (Algorithm* 9*),* (2) *balancing the blocks (Algorithm* 8*),* (3) *communication (including related preprocessing and postprocessing) during balance refinement,* (4) *local processing during construction,* (5) `BlockPartition`*, and* (6) `Sample Sort`*.*

- We demonstrated scalability up to 1024 processors: we were able to construct and balance octrees with over 1 billion octants in less than one minute.

We need to consider the following factors to improve the performance of the proposed algorithms. In order to minimize communication costs, it is desirable to have as large coarse blocks as possible since the communication cost is proportional to the area of the interprocessor boundaries. However, too coarse blocks will increase the work for the local block balancing stage (section 3.3.2). If additional local splits are introduced, then the intrablock boundaries increase, causing the work load for the first ripple balance to increase. The local balancing step of the algorithm can be made more efficient by performing the local balancing recursively by estimating the correct size of the block that can be balanced by the search-free approach. Such an approach should be based on low-level architecture details, like the cache size.

## Appendix A. Properties of Morton encoding.

PROPERTY 2. *Sorting all the leaves in the ascending order of their Morton ids is identical to a preorder traversal of the leaves of the octree. If one connects the centers of the leaves in this order, one can observe a Z-pattern in the Cartesian space. The space-filling Z-order curve has the property that spatially nearby octants tend to be clustered together. The octants in Figures* 1(b) *and* 1(c) *are all labeled according to this order. Depending on the order of interleaving the coordinates, different Z-order curves are obtained. The two possible Z-curves in two dimensions are shown in the Figure* 19*. Similarly, in three dimensions six different types of Morton ordering are possible.*



| | | 32 | 64 | 128 | 256 | 512 |
|---|---|---|---|---|---|---|
| Construction | ▨ | 59.7 | 9.5 | 6.2 | 3.9 | 4.3 |
| Sort - Construction | ▨ | 2.0 | 0.8 | 0.4 | 0.6 | 1.4 |
| BlockPartition | ▨ | 0.8 | 0.7 | 0.9 | 1.4 | 2.2 |
| Balance | ▥ | 441.8 | 62.8 | 37.5 | 18.8 | 10.8 |
| Commn - Balance | ▨ | 76.4 | 7.5 | 4.7 | 3.1 | 3.8 |
| Internal Balance | ▤ | 335.1 | 40.3 | 23.1 | 9.7 | 3.7 |
| Boundary Balance | ▦ | 30.3 | 15.0 | 9.7 | 5.9 | 3.3 |

FIG. 18. *Fixed-size scalability for a Gaussian distribution of* 128M *octants. 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* 2 *columns, which are further subdivided. The left column is subdivided into* 2 *sections and the right column is subdivided into* 6 *sections. The top and bottom sections of the left column represent the total time taken for* (1) *balance refinement (Algorithm* 11*) and* (2) *construction (Algorithm* 5*), respectively. From top to bottom, the sections of the right column represent the time taken for* (1) *balancing across intra- and interprocessor boundaries (Algorithm* 9*),* (2) *balancing the blocks (Algorithm* 8*),* (3) *communication (including related preprocessing and postprocessing) during balance refinement,* (4) *local processing during construction,* (5) `BlockPartition`*, and* (6) `Sample Sort`.

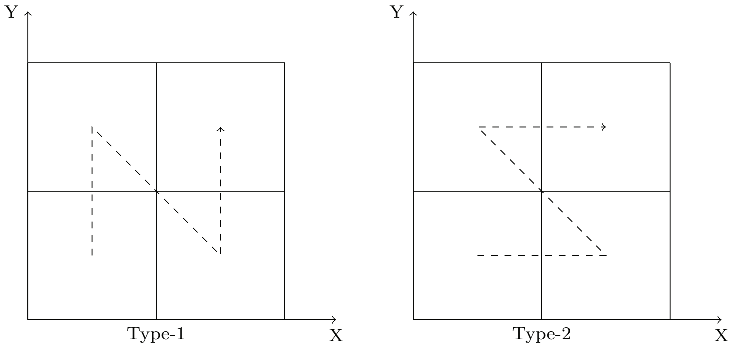


FIG. 19. *Two types of Z-ordering in quadtrees.*

Algorithm 12. Finding the lesser of two Morton ids (sequential).

**Input:** **Two Morton ids,** $A$ **and** $B$ **with different anchors.**
**Output:** $R$, **the lesser of the two Morton ids.**

```
$X_i \leftarrow (A_i \oplus B_i)$, $i \in \{x, y, z\}$
$e \leftarrow \arg\max_i (\lfloor \log_2(X_i) \rfloor)$
if $A_e < B_e$
    $R \leftarrow A$
else
    $R \leftarrow B$
end if
```

Property 3. *Given three octants,* $a < b < c$ *and* $c \notin \{\mathcal{D}(b)\}$,

$$a < d < c \quad \forall d \in \{\mathcal{D}(b)\}.$$

Property 4. *The Morton id of any node is less than those of its descendants.*

Property 5. *Two distinct octants overlap if and only if one is an ancestor of the other.*

Property 6. *The Morton id of any node and of its first child*[24] *are consecutive. It follows from Property* 4 *that the first child is also the child with the least Morton id.*

Property 7. *The first descendant at level* $l$*, denoted by* $\mathcal{FD}(N, l)$*, of any node* $N$ *is the descendant at level* $l$ *with the least Morton id. This can be arrived at by following the first child at every level starting from* $N$*.* $\mathcal{FD}(N, D_{max})$ *is also the anchor of* $N$ *and is also referred to as the* deepest first descendant*, denoted by* $\mathcal{DFD}(N)$*, of node* $N$*.*

Property 8. *The range* $(N, \mathcal{DFD}(N)]$ *contains only the first descendants of* $N$ *at different levels and hence there can be no more than one leaf in this range in the entire linear octree.*

Property 9. *The* last descendant *at level* $l$*, denoted by* $\mathcal{LD}(N, l)$*, of any node* $N$ *is the descendant at level* $l$ *with the greatest Morton id. This can be arrived at by following the* last child[25] *at every level starting from* $N$*.* $\mathcal{LD}(N, D_{max})$ *is also referred to as the* deepest last descendant*, denoted by* $\mathcal{DLD}(N)$*, of node* $N$*.*

Property 10. *Every octant in the range* $(N, \mathcal{DLD}(N)]$ *is a descendant of* $N$*.*

**Appendix B. Multicomponent Morton representation.** Every Morton id is a set of 4 entities: The three coordinates of the anchor of the octant and the level of the octant. We have implemented the node as a C++ class, which contains these 4 entities as its member data. To use this set as a locational code for octants, we define two primary binary logical operations on it: (a) Comparing if 2 ids are equal and (b) comparing if one id is lesser than the other.

Two ids are equal if and only if all the 4 entities are respectively equal. If two ids have the same anchor, then the one at a coarser level has a lesser Morton id. If the anchors are different, then we can use Algorithm 12 to determine the lesser id. The Z-ordering produced by this operator is identical to that produced by the scalar Morton ids described in section 2.1. The other logical operations can be readily derived from these two operations.

[24] The child that has the same anchor as the parent.

[25] Child with the greatest Morton id.

**Appendix C. Analysis of the block partitioning algorithm.** Assume that the input to the partitioning algorithm is a sorted distributed list of $N$ octants. Then we can guarantee coarsening of the input if there are more than eight octants[26] per processor. The minimum number of octants on any processor, $n_{min}$, can be expressed in terms of $N$ and the imbalance factor,[27] $c$, as follows:

$$n_{min} = \frac{N}{1 + c(n_p - 1)}.$$

This implies that the coarsening algorithm will coarsen the octree if

$$n_{min} = \frac{N}{1 + c(n_p - 1)} > 2^d,$$

$$\implies N > 2^d(1 + c(n_p - 1)).$$

The total number of blocks created by our coarsening algorithm is $\mathcal{O}(p)$. Specifically, the total number of blocks produced by the coarsening algorithm, $N_{blocks}$, satisfies

$$p \leq N_{blocks} < 2^d p.$$

If the input is sorted and if $c \approx 1$, then the communication cost for this partition is $\mathcal{O}(\frac{N}{n_p})$.

**Appendix D. Special case during construction.** We cannot always guarantee the coarsest possible octree for an arbitrary distribution of $N$ points and arbitrary values of $N^p_{max}$, especially when $N^p_{max} \approx \frac{N}{n_p}$. However, if every processor has at least two *well-separated*[28] points and if $N^p_{max} = 1$, then the algorithm will produce the coarsest possible octree under these constraints. However, this is not too restrictive because the input points can always be sampled in such a way that the algorithm produces the desired octree. In addition, the maximum depth of the octree can also be used to control the coarseness of the resulting octree. In all our experiments, we used $N^p_{max} = 1$ and we always got the same octree for different number of processor counts (Table 3).

**Acknowledgments.** The authors thank Santi Swaroop Adavani and Shravan Veerapaneni of the University of Pennsylvania for providing us with an implementation of the parallel `Sample Sort` algorithm.

## REFERENCES

[1] D. Ayala, P. Brunet, R. Juan, and I. Navazo, *Object representation by means of nonminimal division quadtrees and octrees*, ACM Trans. Graph., 4 (1985), pp. 41–59.

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

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

[26] $2^d$ cells for a $d$-tree.

[27] The imbalance factor is the ratio between the maximum and minimum number of octants on any processor.

[28] Convert the points into octants at $D_{max}$ level. If there exists at least one coarse octant between these two octants, then the points are considered to be well separated.

[4] M. W. Bern, D. Eppstein, and S.-H. Teng, *Parallel construction of quadtrees and quality triangulations*, Internat. J. Comput. Geom. Appl., 9 (1999), pp. 517–532.

[5] G. Berti, *Image-based unstructured* 3*-d mesh generation for medical applications*, in European Congress on Computational Methods in Applied Sciences and Engineering, Jyväskylä, Finland, 2004.

[6] P. Brunet and I. Navazo, *Solid representation and operation using extended octrees*, ACM Trans. Graph., 9 (1990), pp. 170–197.

[7] 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, Department of Computer Science, Williams College, 2003.

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

[9] R. A. Finkel and J. L. Bentley, *Quad trees: A data structure for retrieval on composite keys*, Acta Inform., 4 (1974), pp. 1–9.

[10] L. A. Freitag and R. M. Loy, *Adaptive, multiresolution visualization of large data sets using a distributed memory octree*, in ACM/IEEE 1999 Conference on Supercomputing, 1999.

[11] S. Frisken and R. Perry, *Simple and efficient traversal methods for quadtrees and octrees*, J. Graphics Tools, 7 (2002), pp. 1–11.

[12] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *An Introduction to Parallel Computing: Design and Analysis of Algorithms*, 2nd ed., Addison-Wesley, Reading, MA, 2003.

[13] D. M. Greaves and A. G. L. Borthwick, *Hierarchical tree-based finite element mesh generation*, Internat. J. Numer. Methods Engrg., 45 (1999), pp. 447–471.

[14] 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, Vol. 12, Bonn, Germany, 1997, E. H. D'Hollander, G. R. Joubert, F. J. Peters, and U. Trottenberg, eds., Elsevier, North-Holland, Amsterdam, 1998, pp. 589–600.

[15] B. Hariharan, S. Aluru, and B. 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, IEEE Computer Society Press, Los Alamitos, CA, 2002, pp. 1–17.

[16] B. Von Herzen and A. H. Barr, *Accurate triangulations of deformed, intersecting surfaces*, in SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, ACM Press, New York, 1987, pp. 103–110.

[17] 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, American Geophysical Union, Washington, DC, 2002.

[18] H. Magistrale, S. Day, R. Clayton, and R. Graves, *The SCEC Southern California reference three-dimensional seismic velocity model version* 2, Bull. Seismological Soc. Amer., 90 (2000), pp. S65–S76.

[19] D. Meagher, *Geometric modeling using octree encoding*, Comput. Graphics Image Process., 19 (1982), pp. 129–147.

[20] D. Moore, *The cost of balancing generalized quadtrees*, in Symposium on Solid Modeling and Applications, 1995, pp. 305–312.

[21] S. Narasimhan, R.-P. Mundani, and H.-J. Bungartz, *An octree and a graph-based approach to support location aware navigation services*, in Proceedings of the 2006 International Conference on Pervasive Systems & Computing (PSC 2006), CSREA Press, Las Vegas, NV, 2006, pp. 24–30.

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

[23] H. Samet, *The quadtree and related hierarchical data structures*, ACM Comput. Surv., 16 (1984), pp. 187–260.

[24] R. Schneiders, *An algorithm for the generation of hexahedral element meshes based on an octree technique*, in Proceedings of the 6th International Meshing Roundtable, 1997, pp. 183–194.

[25] R. Schneiders, R. Schindler, and F. Weiler, *Octree-based generation of hexahedral element meshes*, in Proceedings of the 5th International Meshing Roundtable (Pittsburgh), 1996, pp. 205–216.

[26] M. S. Shephard and M. K. Georges, *Automatic three-dimensional mesh generation by the finite octree technique*, Internat. J. Numer. Methods Engrg., 26 (1991), pp. 709–749.

[27] J. R. Shewchuk, *Tetrahedral mesh generation by Delaunay refinement*, in Proceedings of the Fourteenth Annual Symposium on Computational Geometry (Minneapolis, MN), ACM,

New York, 1998, pp. 86–95.

[28] A. K. Somani and A. M. Sansano, *Minimizing Overhead in Parallel Algorithms through Overlapping Communication/Computation*, Tech. report, Institute for Computer Applications in Science and Engineering (ICASE), 1997.

[29] K. C. Strasters and J. J. Gerbrands, 3-*dimensional image segmentation using a split, merge and group-approach*, Pattern Recognition Lett., 12 (1991), pp. 307–325.

[30] S.-H. Teng, *Provably good partitioning and load-balancing algorithms for parallel adaptive N-body simulation*, SIAM J. Sci. Comput., 19 (1998), pp. 635–656.

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

[32] T. Tu and D. R. O'Hallaron, *Balance Refinement of Massive Linear Octree Datasets*, Technical Report CMU-CS-04, Carnegie Mellon University, Pittsburgh, 2004.

[33] T. Tu and D. R. O'Hallaron, *Extracting hexahedral mesh structures from balanced linear octrees*, in Proceedings of the 13th International Meshing Roundtable (Williamsburg, VA), Sandia National Laboratories, 2004, pp. 191–200.

[34] 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, IEEE Computer Society Press, Washington, DC, 2005, p. 4.

[35] M. S. Warren and J. K. Salmon, *Astrophysical N-body simulations using hierarchical tree data structures*, in Supercomputing '92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 570–576.

[36] M. S. Warren and J. K. Salmon, *A parallel hashed octree N-body algorithm*, in Proceedings of Supercomputing '93, 1993, pp. 12–21.

[37] L. Ying, G. Biros, and D. Zorin, *A kernel-independent adaptive fast multipole algorithm in two and three dimensions*, J. Comput. Phys., 196 (2004), pp. 591–626.

[38] L. Ying, G. Biros, D. Zorin, and H. Langston, *A new parallel kernel-independent fast multipole method*, in SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, IEEE Computer Society, Washington, DC, 2003, p. 14.