

Calhoun: The NPS Institutional Archive

# Recursive algorithms for distributed forests of octrees 

Isaac, Tobin

http://hdl.handle.net/10945/45450


Calhoun is a project of the Dudley Knox Library at NPS, furthering the precepts and goals of open government and government transparency. All information contained herein has been approved for release by the NPS Public Affairs Officer.

Dudley Knox Library / Naval Postgraduate School 411 Dyer Road / 1 University Circle Monterey, California USA 93943

# RECURSIVE ALGORITHMS FOR DISTRIBUTED FORESTS OF OCTREES 

TOBIN ISAAC＊，CARSTEN BURSTEDDE ${ }^{\dagger}$ ，LUCAS C．WILCOX＊キ，AND OMAR GHATTAS＊§ ${ }^{\text {『 }}$


#### Abstract

The forest－of－octrees approach to parallel adaptive mesh refinement and coarsening （AMR）has recently been demonstrated in the context of a number of large－scale PDE－based appli－ cations．Efficient reference software has been made freely available to the public both in the form of the standalone p4est library and more indirectly by the general－purpose finite element library deal．II，which has been equipped with a p4est backend．

Although linear octrees，which store only leaf octants，have an underlying tree structure by definition，it is not fully exploited in previously published mesh－related algorithms．This is because the branches are not explicitly stored，and because the topological relationships in meshes，such as the adjacency between cells，introduce dependencies that do not respect the octree hierarchy．In this work we combine hierarchical and topological relationships between octants to design efficient recursive algorithms that operate on distributed forests of octrees．

We present three important algorithms with recursive implementations．The first is a parallel search for leaves matching any of a set of multiple search criteria，such as leaves that contain points or intersect polytopes．The second is a ghost layer construction algorithm that handles arbitrarily refined octrees that are not covered by previous algorithms，which require a $2: 1$ condition between neighboring leaves．The third is a universal mesh topology iterator．This iterator visits every cell in a partition，as well as every interface（face，edge and corner）between these cells．The iterator calculates the local topological information for every interface that it visits，taking into account the nonconforming interfaces that increase the complexity of describing the local topology．To demonstrate the utility of the topology iterator，we use it to compute the numbering and encoding of higher－order $C^{0}$ nodal basis functions used for finite elements．

We analyze the complexity of the new recursive algorithms theoretically，and assess their perfor－ mance，both in terms of single－processor efficiency and in terms of parallel scalability，demonstrating good weak and strong scaling up to 458 k cores of the JUQUEEN supercomputer．


Key words．forest of octrees，parallel adaptive mesh refinement，Morton code，recursive algo－ rithms，large－scale scientific computing

AMS subject classifications． $65 \mathrm{M} 50,68 \mathrm{~W} 10,65 \mathrm{Y} 05,65 \mathrm{D} 18$

1．Introduction．The development of efficient and scalable parallel algorithms that modify computational meshes is necessary for resolving features in large－scale simulations．These features may vanish and reappear，and／or evolve in shape and location，which stresses the dynamic and in－situ aspects of adaptive mesh refinement and coarsening（AMR）．Both stationary and time－dependent simulations benefit from flexible and fast remeshing and repartitioning capabilities，for example when using a－ posteriori error estimation，building mesh hierarchies for multilevel solvers for partial differential equations（PDEs），or tracking of non－uniformly distributed particles by using an underlying adaptive mesh．

Three main algorithmic approaches to AMR have emerged over time，which we may call unstructured（ U ），block－structured（S），and hierarchical or tree－based（T） AMR．Just some examples that integrate parallel processing are（U） $16[21$ ，（S） 7.12 ， 17．18］，and（T）25 27．While these approaches have been developed independently

[^0]of one another, there has been a definite crossover of key technologies. The graphbased partitioning algorithms traditionally used in UAMR have for instance been supplemented by fast algorithms based on coordinate partitioning and space-filling curves (SFCs) [9] Hierarchical ideas and SFCs have also been applied in SAMR packages to speed up and improve the partitioning 8, 10]. Last but not least, the unstructured meshing paradigm can be employed to create a root mesh of connected trees when a nontrivial geometry needs to be meshed by forest-of-octrees TAMR 6.

The three approaches mentioned above differ in the way that the mesh topology information is passed to applications. With UAMR, the mesh is usually stored in memory as an adjacency graph, and the application traverses the graph to compute residuals, assemble system matrices, etc. This approach has the advantages that local graph traversal operations typically have constant runtime complexity and that the AMR library can remain oblivious of the details of the application, but the disadvantages of less efficient global operations, such as locating the cell containing a point, and of unpredictable memory access. On the other hand, the SAMR approach allows for common operations to be optimized and to use regular memory access patterns, but requires more integration between the AMR package and the application, which may not have access to the topology in a way not anticipated by the AMR package.

Tree-based AMR can be integrated with an application for convenience [23], but can also be kept strictly modular 27]. Most TAMR packages implement logarithmiccomplexity algorithms for both global operations, such as point location, and local operations, such as adjacency queries. The paper 6] introduces the p4est library, which implements distributed forest-of-octree AMR with an emphasis on geometric and topological flexibility and parallel scalability, and connects with applications through a minimal interface.

The implementation of p4est does not explicitly build a tree data structure, so tree-based, recursive algorithms are largely absent from the original presentation in 6]. Many topological operations on octrees and quadtrees, however, are naturally expressed as recursive algorithms, which have simple descriptions and often have good, cache-oblivious memory access patterns. In this paper, we present, analyze, and demonstrate the efficiency of algorithms for important hierarchical and topological operations: searching for leaves matching multiple criteria in parallel, identifying neighboring domains from minimal information, and iterating over mesh cells and interfaces. Each algorithm has a key recursive component that gives it an advantage over previously developed non-recursive algorithms, such as improved efficiency, coverage of additional use cases, or both. We demonstrate the per-process efficiency of these algorithms, as well as their parallel scalability on JUQUEEN 15], a Blue Gene/Q 13 supercomputer.
2. Forest of octree types and operations. Here we present the important concepts on which we build our algorithms. We review the data structures for octants and distributed forests of octrees that were presented in [6]. We also define a data type to handle both octants and octant boundaries that will allow us to describe the topology of forests of octrees ${ }^{1]}$ The definitions in this section are summarized in Table 2.1. For the sake of correctness, the definitions in this section are given formally, but the reader may find that the figures are just as helpful in understanding

[^1]TABLE 2.1
A summary of Section 2 and the locations of the definitions in the text.

| § 2.1 Octants and points |  |  |
| :---: | :---: | :---: |
| ```octant o, dom(o) root}(t atom a {\mp@subsup{\operatorname{dom}}{b}{}(o)\mp@subsup{}}{b\in\mathcal{B}\cup{\mp@subsup{v}{0}{}}}{} point c = (o,b) dom(c) dim(c) level(c)``` | Data type and the cube in $\mathbb{R}^{d}$ it represents Root of the $t$-th octree: side length $2^{l_{\text {max }}}$ level- $l_{\text {max }}$ octant: side length 1 Octant boundary domains and their indices Common data type for octants and interfaces A point's $n$-dim. $(n \leq d)$ hypercube domain Topological dimension of a point Refinement level for points | $2$ <br> Fig. 2.1 <br> 2.3) <br> (2.4) |
| $\S 2.2$ Hierarchical and topological relationships |  |  |
| $\operatorname{desc}(c)$ $\operatorname{child}(c)$ $\operatorname{part}(c)$ $\operatorname{clos}(c)$ $\operatorname{bound}(c)$ $\operatorname{supp}(c)$ $\operatorname{atom} \operatorname{supp}(c)$ |  | (2.5 ${ }_{(2.6}^{\overline{2.7}}$ |
| $\S 2.3$ Forests of octrees |  |  |
| $\mathcal{T}:=\left\{\left(T^{t}, \varphi^{t}\right)\right\}_{0 \leq t<K}$ <br> Dom(c) $\mathcal{O}:=\bigsqcup_{0 \leq t<K} \mathcal{O}^{t}$ | Conformal macro mesh of $\Omega$ <br> Point domain mapped by $\varphi^{t}$ into $\Omega$ <br> Non-conformal mesh via octree refinement | (2.13 |
| § 2.4 Distributed forests of octrees |  |  |
| $\begin{aligned} & \mathrm{o} \leq \mathrm{r} \\ & \mathcal{O}_{\mathfrak{p}}:=\bigsqcup_{0 \leq t<K} \mathbf{O}_{\mathfrak{p}}^{t} \\ & \Omega_{\mathfrak{p}}, \Omega^{t}, \Omega_{\mathfrak{p}}^{t} \\ & \text { locate(a) } \\ & \left(\mathrm{f}_{\mathfrak{q}}, 1_{\mathfrak{q}}\right):=\operatorname{range}(\mathfrak{q}) \\ & \left(\mathrm{f}_{\mathfrak{o}}, \mathrm{l}_{\mathfrak{o}}\right):=\operatorname{range}(\mathrm{o}) \\ & \mathbf{f}:=\left\{\mathrm{f}_{\mathfrak{q}}\right\}_{0 \leq \mathfrak{q}<P} \\ & \mathrm{~F}_{\mathfrak{p}}:=\left(\mathcal{T}, \mathcal{O}_{\mathfrak{p}}, \mathbf{f}\right) \end{aligned}$ | SFC-based total octant order <br> Sorted arrays of leaves owned by $\mathfrak{p}$ for each tree Subdomains of $\mathcal{O}_{\mathfrak{p}}, \mathcal{O}^{t}, \mathbf{O}_{\mathfrak{p}}^{t}$ <br> Process $\mathfrak{q}$ such that $\operatorname{Dom}(\mathrm{a}) \subseteq \Omega_{\mathfrak{q}}$ for atom a <br> First and last atoms located in $\Omega_{\mathfrak{q}}$ <br> First and last atoms in octant o's descendants Sorted array of the first atoms of all processes Distributed forest of octrees | Alg. 2.1 <br> 2.16 <br> $\overline{2.17}$ <br> $\underline{\overline{2.18}}$ <br> $\overline{2.19}$ <br> 2.20 <br> $\overline{2.21}$ <br> $\overline{2.22}$ |

our concepts, as they often correspond to geometrically intuitive ideas.
Remark 2.1 (Notation). If we have defined an operation op $(\cdot)$ for every $a \in \mathcal{A}$, then $\operatorname{op}(\mathcal{A}):=\{\operatorname{op}(a): a \in \mathcal{A}\} .|\mathcal{A}|$ is the cardinality of set $\mathcal{A}$. If $\left\{\mathcal{A}_{i}\right\}_{i \in \mathcal{I}}$ are disjoint, their union is written $\bigsqcup_{i \in \mathcal{I}} \mathcal{A}_{i}$. For a subset $A$ of a manifold, $\bar{A}, \partial A$, and $A^{\circ}$ are the closure, boundary, and interior of $A$. We distinguish variable types with fonts:

- standard lower-case for integers and index sets $(a, b, c, \ldots)$, except for $K, N$, and $P$, which are the number of octrees, octants, and processes,
- typewriter for compound data types ( $\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots$ ),
- Fraktur for MPI processes $(\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \ldots)$,
- upper-case for subsets of $\mathbb{R}^{d}$ and manifolds $(A, B, C, \ldots)$,
- calligraphic for finite sets ( $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$ ), and
- bold for finite sets represented as indexable arrays ( $\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$ ).


Fig. 2.1. (left) An illustration of the domains of a root octant r , a level-2 octant o , and an atom a. (right) The correlation between the boundary indices in $\mathcal{B}$ (see the definition at the bottom of this page) and the lower-dimensional hypercubes-squares, line segments, and verticesin the boundary of an octant, with the central cube labeled with the volume index, $v_{0}$ (adapted with permission from [6, Fig. 2]). These indices are used to define points.
2.1. Octants and points. Here we define the octant data type, which we will use in our algorithms, and some special octants, which are illustrated in Figure 2.1.

An octant $(d=3)$ or quadrant $(d=2)$ o has the following data fields:

- o.t $\in \mathbb{N}$-o's tree index, relevant to forests of octrees (see Section 2.3) ${ }^{2}$
- o.l $\in\left\{0,1, \ldots, l_{\max }\right\}-$ o's level of refinement (or just level);
- $0 . x \in \mathbb{Z}^{d}-\mathrm{o}$ 's coordinates, whose components must be multiples of $2^{l_{\text {max }}-0 . l}$. The fields o.l and o. $x$ encode an open cube in $\mathbb{R}^{d}$ —o's domain-with sides of length $2^{l_{\text {max }}-0 . l}$,

$$
\begin{equation*}
\operatorname{dom}(\mathrm{o}):=\left\{X \in \mathbb{R}^{d}: 0 . x_{i}<X_{i}<0 . x_{i}+2^{l_{\max }-o . l}, 0 \leq i<d\right\} . \tag{2.1}
\end{equation*}
$$

For every tree index $t$, the root is the level- 0 octant whose domain is $\left(0,2^{l_{\max }}\right)^{d}$ :

$$
\begin{equation*}
\operatorname{root}(t) \cdot t:=t, \quad \operatorname{root}(t) \cdot l:=0, \quad \operatorname{root}(t) \cdot x:=(0)^{d} . \tag{2.2}
\end{equation*}
$$

An atom a is a smallest-possible octant, which has a. $l=l_{\max }$ and sides of length 1 .
The algorithms we present involve both the hierarchical aspect of octrees and the topological aspect of their domains. Here we define a data type, which we will call a point, that encompasses both octants and their interfaces. We will then define topological and hierarchical operations for points in Section 2.2. We present these definitions in the context of a single octree. Minor modifications will be necessary for a forest of octrees, which we will discuss in Section 2.3 (see Remark 2.9).

The boundary of a cube in $\mathbb{R}^{d}$ has a standard partition into lower-dimensional hypercubes, which contains $2^{d-n}\binom{d}{n} n$-dimensional hypercubes for $0 \leq n<d$. We index these $\left(3^{d}-1\right)$ hypercubes with a set of boundary indices $\mathcal{B}$. The boundary domain $\operatorname{dom}_{b}(\mathrm{o}), b \in \mathcal{B}$, is the corresponding hypercube in the boundary of dom(o). For $d=3, \mathcal{B}$ is made of eight corner indices $\left\{c_{i}\right\}_{0 \leq i<8}$, twelve edge indices $\left\{e_{i}\right\}_{0 \leq i<12}$, and six face indices $\left\{f_{i}\right\}_{0 \leq i<6}$, which are all illustrated in Figure 2.1. For convenience, we define one additional index, the volume index $v_{0}$ corresponding to the volume of an octant, which defines an alias of an octant's domain, $\operatorname{dom}_{v_{0}}(\mathrm{o}):=\operatorname{dom}(\mathrm{o})$.

[^2]A point is a tuple $(o, b)$, where $o$ is an octant and $b \in \mathcal{B} \cup\left\{v_{0}\right\}$. The domain of $a$ point $\mathrm{c}=(\mathrm{o}, b)$ is

$$
\begin{equation*}
\operatorname{dom}(\mathrm{c}):=\operatorname{dom}_{b}(\mathrm{o}) \tag{2.3}
\end{equation*}
$$

Two points are equal if and only if their domains are equal. The dimension $\operatorname{dim}(\mathrm{c})$ of a point c is the topological dimension of its domain. If $\operatorname{dim}(\mathrm{c})=n, \mathrm{c}$ is an $n$-point. The level of a point c is the minimum refinement level of all octants in the vicinity of $c$ that appear in a point tuple equal to $c$,

$$
\begin{equation*}
\operatorname{level}(\mathrm{c}):=\min \left\{o . l: \exists b \in \mathcal{B} \cup\left\{v_{0}\right\},(\mathrm{o}, b)=\mathrm{c}\right\} \tag{2.4}
\end{equation*}
$$

If $\operatorname{dim}(c)>0$, there is no need to use the minimum refinement level in the definition of level(c), because all octants that have c as a boundary point have the same refinement level: $(c=(o, b)) \Leftrightarrow(\operatorname{level}(c)=o . l)$. A 0-point, however, may be a corner point for octants with different refinement levels: by choosing the minimum we make a 0 -point's level match that of the biggest neighboring octant.

REMARK 2.2. We consider octants to be points: when an operation on a point is applied to an octant o , one should understand $\left(\mathrm{o}, v_{0}\right)$.
2.2. Hierarchical and topological relationships. Here we define the hierarchical and topological relationships used in our algorithms and proofs below. We show what these relationships look like in Table 2.2 .

The hierarchical relationships between points are determined by set inclusion of their domains. The descendants of a point c are all of the points with the same dimension whose domains are contained in the domain of $c$,

$$
\begin{equation*}
\operatorname{desc}(c):=\{e: \operatorname{dim}(e)=\operatorname{dim}(c), \operatorname{dom}(e) \subseteq \operatorname{dom}(c)\} \tag{2.5}
\end{equation*}
$$

The children of a point c are descendants that are more refined than c by one level,

$$
\begin{equation*}
\operatorname{child}(\mathrm{c}):=\{\mathrm{h}: \mathrm{h} \in \operatorname{desc}(\mathrm{c}), \operatorname{level}(\mathrm{h})=\operatorname{level}(\mathrm{c})+1\} \tag{2.6}
\end{equation*}
$$

The requirement that an octant o's coordinates must be multiples of $2^{l_{\max }-o . l}$ has the consequence that the domains of two distinct points with the same level do not overlap, and that every point's domain is tiled by the domains of its children (a collection $\mathcal{U}$ of subsets of a set $S$ in a topological space tiles $S$ if $S \subseteq \bigcup_{U \in \mathcal{U}} \bar{U}$ and $(U, V \in$ $\mathcal{U}, U \neq V) \Rightarrow(U \cap V=\emptyset))$.

Proposition 2.3. If $\operatorname{dim}(c)>0$, then $|\operatorname{child}(c)|=2^{\operatorname{dim}(c)}$ and $\operatorname{dom}(\operatorname{child}(c))$ tiles $\operatorname{dom}(\mathrm{c})$.

A point's domain is tiled by its children's domains, but it is not partitioned by them. To define a partition, we must add lower-dimensional points between them. The child partition is the set of all points whose domains are contained in dom(c) and whose levels are greater by one,

$$
\begin{equation*}
\operatorname{part}(\mathrm{c}):=\{\mathrm{h}: \operatorname{level}(\mathrm{h})=\operatorname{level}(\mathrm{c})+1, \operatorname{dom}(\mathrm{~h}) \subset \operatorname{dom}(\mathrm{c})\} \tag{2.7}
\end{equation*}
$$

Proposition 2.4. If $\operatorname{dim}(c)>0$, then $|\operatorname{part}(\mathrm{c})|=3^{\operatorname{dim}(\mathrm{c})}$ and $\operatorname{part}(\mathrm{c})$ defines $a$ partition, $\operatorname{dom}(c)=\bigsqcup \operatorname{dom}(\operatorname{part}(c))$.

The two basic topological sets we need for a point c are the lower-dimensional points that surround c-its boundary points-and the octants that surround c-its

Table 2.2
For $d=3$, illustrations of the boundary sets $($ bound $(\mathrm{c}))$, children $($ child $(\mathrm{c}))$, child partition sets (part(c)), and support sets $(\operatorname{supp}(c))$ of octants and lower-dimensional n-points. The closure set, not illustrated, is the union of the point with its boundary set. For a 0-point, the atomic support set atomsupp(c) looks like the support set, only scaled down.

| $n$ | c | bound(c) | child(c) | part(c) | $\operatorname{supp}(\mathrm{c})$ |
| :---: | :---: | :---: | :---: | :---: | :---: |
| 0 | (corner) | $\emptyset$ | $\emptyset$ | $\emptyset$ |  |
| 1 | $\mid$ |  | $\begin{aligned} & \text { \| } \\ & \mid \end{aligned}$ |  |  |
| 2 |  |  | $\square$ |  |  |
| 3 |  |  |  |  |  |

support octants. To define boundary points, we first define closure points. The closure set of an octant $\circ$ is the set of all points in which o may appear in a point tuple,

$$
\begin{equation*}
\operatorname{clos}(\mathrm{o}):=\left\{(\mathrm{o}, b): b \in \mathcal{B} \cup\left\{v_{0}\right\}\right\} \tag{2.8}
\end{equation*}
$$

The closure set of a point c is the intersection of all octant closure sets containing c,

$$
\begin{equation*}
\operatorname{clos}(\mathrm{c}):=\bigcap\{\operatorname{clos}(\mathrm{o}): \mathrm{c} \in \operatorname{clos}(\mathrm{o})\} . \tag{2.9}
\end{equation*}
$$

The boundary set of a point c is its closure less itself,

$$
\begin{equation*}
\text { bound }(c):=\operatorname{clos}(c) \backslash\{c\} . \tag{2.10}
\end{equation*}
$$

Proposition 2.5 (Point closure matches $\mathbb{R}^{d}$ closure). The domains of points in c's closure set clos(c) partition the closure of its domain, $\overline{\operatorname{dom}(c)}=\bigsqcup \operatorname{dom}(\operatorname{clos}(\mathrm{c}))$.

The support set of a point c is the set of octants with the same refinement level as c whose closures include c,

$$
\begin{equation*}
\operatorname{supp}(\mathrm{c}):=\{\mathrm{o}: \mathrm{c} \in \operatorname{clos}(\mathrm{o}), \text { o. } l=\operatorname{level}(\mathrm{c})\} \tag{2.11}
\end{equation*}
$$

Proposition $2.6\left(\mathbb{R}^{d}\right.$ intersection implies support intersection). If $\circ$ is an octant, c is a point, and $\overline{\operatorname{dom}(\mathrm{o})} \cap \operatorname{dom}(\mathrm{c}) \neq \emptyset$, then there exists $\mathrm{s} \in \operatorname{supp}(\mathrm{c})$ such that $\mathbf{s} \in \operatorname{desc}(o)$ or $o \in \operatorname{desc}(s)$.

The following proposition shows the duality between $\operatorname{clos}(\cdot)$ and $\operatorname{supp}(\cdot)$.
Proposition 2.7. If $\operatorname{dim}(c)>0$, then $(o \in \operatorname{supp}(c)) \Leftrightarrow(c \in \operatorname{clos}(o))$.
For 0-points, this duality does not hold because a 0 -point can be in the closure set of an octant with a more refined level. In fact, every 0-point is in the closure of an atom. The atomic support set of a 0-point c is the set of atoms whose closures include c,

$$
\begin{equation*}
\operatorname{atom} \operatorname{supp}(\mathrm{c}):=\left\{\mathrm{a}: \mathrm{c} \in \operatorname{clos}(\mathrm{a}), \text { a. } l=l_{\max }\right\} \tag{2.12}
\end{equation*}
$$

The support and atomic support sets of a 0-point contain and are contained in all neighboring octants, respectively.

Proposition 2.8. If $\operatorname{dim}(\mathrm{c})=0$, o is a octant, and $\mathrm{c} \in \operatorname{clos}(\mathrm{o})$, then there are $\mathrm{a} \in \operatorname{atom} \operatorname{supp}(\mathrm{c})$ and $\mathrm{s} \in \operatorname{supp}(\mathrm{c})$ such that $\mathrm{a} \in \operatorname{desc}(\mathrm{o})$ and $\mathrm{o} \in \operatorname{desc}(\mathrm{s})$.
2.3. Forests of octrees. A forest of quadtrees $(d=2)$ or octrees $(d=3)$ is a mesh of a $d$-dimensional domain $\Omega$ with two layers, a macro layer and a micro layer. The macro layer is a geometrically conformal mesh ${ }^{3}$ of $K$ mapped cells (quadrilaterals $(d=2)$ or hexahedra $(d=3))$,

$$
\begin{equation*}
\mathcal{T}:=\left\{\left(T^{t}, \varphi^{t}\right)\right\}_{0 \leq t<K}, \tag{2.13}
\end{equation*}
$$

where each $T^{t}$ has an associated map $\varphi^{t}: \overline{\operatorname{dom}(\operatorname{root}(t))} \rightarrow \overline{T^{t}}$, which is a continuous bijection between the domain of the root octant and $T^{t}$. We define the mapped domain of a point $\mathrm{c}=(\mathrm{o}, b)$ by its image under the map for o's tree index,

$$
\begin{equation*}
\operatorname{Dom}(c):=\varphi^{o . t}\left(\operatorname{dom}_{b}(o)\right) . \tag{2.14}
\end{equation*}
$$

Remark 2.9 (Modifications to definitions for forests). In Section 2.1 we defined points in the context of a single octree. In the forest-of-octrees context, we consider two points equal if their mapped domains are equal. If one substitutes mapped domains for unmapped domains in the definitions and propositions in Section 2.2, they hold in the forest-of-octrees context. If a point c's mapped domain is on the boundary between macro-layer cells, then c's support set supp(c) no longer has the regular shape shown in Table 2.2, but depends on the macro layer topology. A face on the boundary of $\Omega$, for example, has only one support octant. We emphasize that in the forest-of-octrees context, the sets defined in Section 2.2 do not depend on the exact nature of the maps $\left\{\varphi^{t}\right\}_{0 \leq t<K}$, but can be constructed, in time proportional to their sizes, from the point data of their arguments and the mesh topology of $\mathcal{T}$, i.e., which cells are neighbors, which of their faces correspond, and how those faces are oriented relative to each other. These issues are covered in 6, Section 2.2].

For each $0 \leq t<K$, the tree-t leaves $\mathcal{O}^{t} \subset \operatorname{desc}(\operatorname{root}(t))$ are a set of $N^{t}$ octants whose domains tile $\operatorname{dom}(\operatorname{root}(t))$. The micro layer $\mathcal{O}$ is the union of these sets, and its size is $N$,

$$
\begin{equation*}
\mathcal{O}:=\bigsqcup_{0 \leq t<K} \mathcal{O}^{t}, N:=|\mathcal{O}|=\sum_{0 \leq t<K} N^{t} \tag{2.15}
\end{equation*}
$$

[^3]

Fig. 2.2. (Adapted with permission from [6, Fig. 2.1].) A $d=2$ example of the relationship between the implicit tree structure (left) and the domain tiling (right) of a forest of octrees with two trees $s$ and $t$. The bijections $\varphi^{s}$ and $\varphi^{t}$ map the implicit coordinate systems of the unmapped octant domains onto the cells $T^{s}$ and $T^{t}$. The left-to-right traversal of the leaves (black "zig-zag" line) demonstrates the total order (Algorithm 2.1). In this example the forest is partitioned among three processes $\mathfrak{p}, \mathfrak{q}$ and $\mathfrak{r}$ by sectioning the leaves into $\mathcal{O}_{\mathfrak{p}}, \mathcal{O}_{\mathfrak{q}}$, and $\mathcal{O}_{r}$. Color conveys this partition, while the node shapes convey the division of the leaves into the trees $\mathcal{O}^{\mathbf{s}}$ and $\mathcal{O}^{t}$. The small, brightly colored nodes represent the first atoms located in each process's subdomains 2.21 .

The mapped domains of the micro layer octants tile $\Omega$, but this tiling is not a geometrically conforming mesh: when neighboring octants have different levels, their faces (and edges if $d=3$ ) do not conform to each other. If neighboring octants differ by at most one level, the forest is said to satisfy a $2: 1$ balance condition 14.23 .

We call $\mathcal{O}^{t}$ the leaves of octree $t$ because one could build a tree structure, starting with $\operatorname{root}(t)$ and using the child $(\cdot)$ operation, whose leaves would be $\mathcal{O}^{t}$. The p4est library does not store this tree structure in memory. Storing just $\mathcal{O}^{t}$ is an approach known as a linear octree representation 26 .
2.4. Distributed forests of octrees. In p4est, the macro layer $\mathcal{T}$ is static and replicated on each process, while the micro layer $\mathcal{O}$ is dynamic-it may be adaptively refined, coarsened, and repartitioned frequently over the life of a forest-and distributed, with each processes owning a distinct subset of leaves. We describe the distribution method here, and illustrate it in Figure 2.2.

The partitioning of leaves between processes and the layout of leaves in memory is determined by the total order induced by the comparison operation in Algorithm 2.1 . The comparison of coordinates in line 2 is defined by a space-filling curve; the p4est library uses the so-called $z$-ordering which corresponds to the Morton curve 20].

```
Algorithm 2.1: \(0 \leq r\) (octant 0 , octant r )
    if o. \(t \neq\) r.t then return ( \(0 . t \leq \mathrm{r} . t\) ) [If \(s<t\), all of octree \(s\) 's leaves come before \(t\) 's.]
    if \(0 . x \neq \mathrm{r} . x\) then return \((0 . x \leq \mathrm{r} . x) \quad\) [Coordinates are ordered by \(S F C\) index.]
    return \((\mathrm{o} . l \leq \mathrm{r} . l) \quad\) [Ancestors precede descendants (preordering).]
```

REmARK 2.10. When we index a sorted array of octants ( $\mathbf{A}[i]$ ), children $(\operatorname{child}(o)[i])$, or a support set $(\operatorname{supp}(c)[i])$, we mean the $i$ th octant with respect to the total order.

Using this total order, each process is assigned a contiguous (with respect to the total order) section of leaves in MPI-rank order. For each $0 \leq t<K$ and $0 \leq \mathfrak{p}<P$, the subset of $\mathcal{O}^{t}$ assigned to process $\mathfrak{p}$ is in an array $\mathbf{O}_{\mathfrak{p}}^{t}$, which has size $N_{\mathfrak{p}}^{t}$; these
arrays partition the tree- $t$ leaves, $\mathcal{O}^{t}=\bigsqcup_{0 \leq \mathfrak{p}<P} \mathbf{O}_{\mathfrak{p}}^{t}$. The set of all leaves assigned to $\mathfrak{p}$ is

$$
\begin{equation*}
\mathcal{O}_{\mathfrak{p}}:=\bigsqcup_{0 \leq t<K} \mathbf{O}_{\mathfrak{p}}^{t} ; \tag{2.16}
\end{equation*}
$$

its size is $N_{\mathfrak{p}}:=\left|\mathcal{O}_{\mathfrak{p}}\right|=\sum_{0 \leq t<K} N_{\mathfrak{p}}^{t}>0 \bigsqcup^{4}$ The subdomain of $\Omega$ tiled by $\operatorname{Dom}\left(\mathbf{O}_{\mathfrak{p}}^{t}\right)$ is the interior of its closure,

$$
\begin{equation*}
\Omega_{\mathfrak{p}}^{t}:=\left(\overline{\bigcup \operatorname{Dom}\left(\mathbf{O}_{\mathfrak{p}}^{t}\right)}\right)^{\circ} \tag{2.17}
\end{equation*}
$$

the subdomains $\Omega^{t}$ and $\Omega_{\mathfrak{p}}$ are analogously defined.
Because the leaves are partitioned, process $\mathfrak{p}$ cannot determine (without communication) if an octant o, whose mapped domain $\operatorname{Dom}(o)$ is outside $\Omega_{\mathfrak{p}}$, is a leaf. We do, however, want $\mathfrak{p}$ to be able to locate the subdomains that overlap Dom(o). To allow this, each process in p4est has some information about other processes' subdomains, which we describe here. We start from the fact that an atom's mapped domain is located in the subdomain of only one process's subdomain,

$$
\begin{equation*}
(\operatorname{locate}(\mathrm{a}):=\mathfrak{q}) \Leftrightarrow\left(\operatorname{Dom}(\mathrm{a}) \subseteq \Omega_{\mathfrak{q}}\right) \tag{2.18}
\end{equation*}
$$

Note that $\mathfrak{q}=$ locate $(\mathrm{a})$ does not imply $\mathrm{a} \in \mathcal{O}_{\mathfrak{q}}$ : a could be a descendant of a leaf in $\mathcal{O}_{\mathfrak{q}}$. To test whether $\mathfrak{q}=$ locate(a) for an arbitrary atom $a$, it is only necessary to precompute a process's range: a tuple of its first atom $\mathbf{f}_{\mathfrak{q}}$ and its last atom $\mathbf{1}_{\mathfrak{q}}$ with respect to the total order of octants,

$$
\begin{align*}
\mathrm{f}_{\mathfrak{q}} & :=\min \left\{\mathrm{a}: \mathrm{a} \cdot l=l_{\max }, \operatorname{Dom}(\mathrm{a}) \subseteq \Omega_{\mathfrak{q}}\right\} \\
\mathrm{l}_{\mathfrak{q}} & :=\max \left\{\mathrm{a}: \mathrm{a} \cdot l=l_{\max }, \operatorname{Dom}(\mathrm{a}) \subseteq \Omega_{\mathfrak{q}}\right\},  \tag{2.19}\\
\operatorname{range}(\mathfrak{q}) & :=\left(\mathrm{f}_{\mathfrak{q}}, \mathrm{l}_{\mathfrak{q}}\right) .
\end{align*}
$$

Proposition 2.11. $(\mathfrak{q}=\operatorname{locate}(\mathrm{a})) \Leftrightarrow\left(\mathrm{f}_{\mathfrak{q}} \leq \mathrm{a} \leq 1_{\mathfrak{q}}\right)$.
Proposition 2.12 (A range describes a subdomain). $\overline{\Omega_{\mathfrak{q}}}=\overline{\operatorname{Dom}\left(\left[\mathbf{f}_{\mathfrak{q}}, \mathbf{1}_{\mathfrak{q}}\right]\right)}$, where $\left[\mathrm{f}_{\mathfrak{q}}, 1_{\mathfrak{q}}\right]$ is the set of all atoms a such that $\mathrm{f}_{\mathfrak{q}} \leq \mathrm{a} \leq \mathbf{1}_{\mathfrak{q}}$.

We also apply the range operator to individual octants: the range of an octant $\circ$ is a tuple of the first and last atoms, $f_{0}$ and $I_{o}$, in its descendants,

$$
\begin{align*}
\mathrm{f}_{\mathrm{o}} & :=\min \left\{\mathrm{a}: \mathrm{a} \cdot l=l_{\max }, \mathrm{a} \in \operatorname{desc}(\mathrm{o})\right\} \\
\mathrm{l}_{\mathrm{o}} & :=\max \left\{\mathrm{a}: \mathrm{a} \cdot l=l_{\max }, \mathrm{a} \in \operatorname{desc}(\mathrm{o})\right\}  \tag{2.20}\\
\text { range }(\mathrm{o}) & :=\left(\mathrm{f}_{\mathrm{o}}, \mathrm{l}_{\mathrm{o}}\right)
\end{align*}
$$

Proposition 2.13. $\left(\operatorname{Dom}(o) \subseteq \Omega_{\mathfrak{q}}\right) \Leftrightarrow\left(\mathfrak{q}=\operatorname{locate}\left(f_{\circ}\right)=\operatorname{locate}\left(l_{o}\right)\right)$.
To locate atoms, it is not necessary to store both $f_{\mathfrak{q}}$ and $l_{\mathfrak{q}}$ : $l_{\mathfrak{q}}$ can be computed from $\mathbf{f}_{\mathfrak{q}+1}$. In p 4 est , we store a sorted array called the first-atoms array $\mathbf{f}$, where

$$
\begin{equation*}
\mathfrak{f}[\mathfrak{p}]:=\mathfrak{f}_{\mathfrak{p}}, 0 \leq \mathfrak{p}<P \tag{2.21}
\end{equation*}
$$

and $\mathbf{f}[P]$ is a phony "terminal" octant whose tree index is $K$. This array is shown in Figure 2.2. The first atom $\mathbf{f}_{\mathfrak{q}}$ is independent of the leaves in $\mathcal{O}_{\mathfrak{q}}$, so $\mathbf{f}$ is up-to-date

[^4]even if other processes have refined or coarsened their leaves. Using f, a process can compute locate(a) in $O(\log P)$ time and test $(\mathfrak{q}=$ locate $(\mathrm{a}) ?)$ in $O(1)$ time.

For the purposes of this paper, we have described all components of a distributed forest of octrees, which is, for process $\mathfrak{p}$, the combination of macro layer (2.13), local leaves (2.16), and the first-atoms array (2.21),

$$
\begin{equation*}
\mathrm{F}_{\mathfrak{p}}:=\left(\mathcal{T}, \mathcal{O}_{\mathfrak{p}}, \mathbf{f}\right) \tag{2.22}
\end{equation*}
$$

REMARK 2.14. $\mathrm{F}_{\mathfrak{p}}$ is an assumed argument of the algorithms we present.
3. Parallel multiple-item search via array splitting. We can optimize the search for a leaf that matches a given condition if we begin at the root of an octree and recursively descend to all children that could possibly be a match. This is a lazy exclusion principle which is motivated by a practical consideration: often an over-optimistic approximate check can be significantly faster than an exact check, which applies to bounding-box checks in computational geometry or to checking the surrounding sphere of a nonlinearly warped octant volume in space.
3.1. Searching in a single octree and in a forest. We assume that the user has a set of arbitrary queries and operations indexed by $\mathcal{Q}$ : for each $q \in \mathcal{Q}, \operatorname{match}_{q}(\cdot)$ returns true or false for every leaf $o \in \mathcal{O}$, and $\mathrm{op}_{q}(o)$ should be executed for every leaf such that $\operatorname{match}_{q}(o)$ is true. Note that this is more general than searching for a single leaf that matches each query: this framework encompasses, for example, the search for all the leaves that intersect a set of polytopes embedded in $\Omega$.

In Search (Algorithm 3.1), we use recursion and lazy exclusion to track multiple simultaneous queries during one traversal. At each recursion into children we only retain the queries that have returned a possible match on the previous level. We implement this by passing as a callback a user-defined lazy matching function Match, which is a boolean operator that takes as arguments an octant o, a boolean isLeaf that indicates if $o \in \mathcal{O}$, and a query index $q \in \mathcal{Q}$ and satisfies the following properties:

- Match (o, isLeaf, $q$ ) returns true if there is a leaf $r \in \mathcal{O}$ that is a descendant of o such that $\operatorname{match}_{q}(r)=$ true, and is allowed to return a false positive (i.e., true even if $\operatorname{match}_{q}(\mathrm{r})$ is false for all descendant leaves of o);
- if isLeaf $=$ true, then Match executes $\mathrm{op}_{q}(\mathrm{o})$ if and only if $\operatorname{match}_{q}(\mathrm{o})=$ true.

```
Algorithm 3.1: Search (octant array A, octant a, index set \(\mathcal{Q}\), callback
Match)
    Input : A is a sorted subset of leaves, \(\mathbf{A} \subseteq \mathcal{O}\);
            a is an ancestor of \(\mathbf{A}[j]\) for each \(j, \mathbf{A} \subseteq \operatorname{desc}(a)\).
    Result \(: \forall q \in \mathcal{Q}, \mathrm{op}_{q}(\mathrm{o})\) is called for every \(\circ \in \mathbf{A}\) such that \(\operatorname{match}_{q}(\mathrm{o})\) is true.
    if \(\mathbf{A}=\emptyset\) then return
    boolean isLeaf \(\leftarrow(\mathbf{A}=\{\mathrm{a}\})\)
    index set \(\mathcal{Q}_{\text {match }} \leftarrow \emptyset \quad\) [queries that pass the lazy criteria at a]
    for all \(q \in \mathcal{Q}\) do
        if Match (a, isLeaf, \(q\) ) then \(\mathcal{Q}_{\text {match }} \leftarrow \mathcal{Q}_{\text {match }} \cup\{q\}\)
    if \(\mathcal{Q}_{\text {match }} \neq \emptyset\) and not isLeaf then
        \(\mathbf{H} \leftarrow\) Split_array \((\mathbf{A}, \mathbf{a} \quad\) [divide \(\mathbf{A}\) between the children of a: see Section 3.2
        for all \(0 \leq i<2^{d}\) do Search \(\left(\mathbf{H}[i]\right.\), \(\operatorname{child}(\mathrm{a})[i], \mathcal{Q}_{\text {match }}\), Match \()\)
```

To extend the action of Search to the whole forest, it can be called once for each tree index $0 \leq t \leq K$ with $\mathbf{O}_{\mathfrak{p}}^{t}$ and $\operatorname{root}(t)$ as arguments. The resulting algorithm
is communication-free and every leaf is queried on only one process, although the ancestors of leaves may appear as arguments to Match for multiple processes.
3.2. Array splitting. Search requires an algorithm Split_array that we have not yet specified. Split_array takes a sorted array of leaves A and an octant a such that each leaf $\mathbf{A}[j]$ is a descendant of a and partitions $\mathbf{A}$ into sorted arrays $\mathbf{H}[0], \mathbf{H}[1]$, $\ldots, \mathbf{H}\left[2^{d}-1\right]$ such that $\mathbf{H}[i]$ contains the descendants of child(a) $[i]$ in $\mathbf{A}$.

Because $\mathbf{A}$ is sorted, the subarrays can be indicated by a non-decreasing sequence of indices $0=\mathbf{k}[0] \leq \mathbf{k}[1] \leq \ldots \leq \mathbf{k}\left[2^{d}\right]=|\mathbf{A}|$, such that $\mathbf{H}[i]=\mathbf{A}[\mathbf{k}[i], \ldots, \mathbf{k}[i+1]-1]$. If child(a) $[i]$ has no descendants in $\mathbf{A}$, this is indicated by $\mathbf{k}[i]=\mathbf{k}[i+1]$.

Let us assume that the children of a have level $l$. If we know that an octant o is a descendant of child(a)[i] for some $i$, then we can compute $i$ from o.x using Algorithm 3.2, which works because we use the Morton order as our space-filling curve. We call this algorithm Ancestor_id, because it is a simple generalization of the algorithm Child_id [6, Algorithm 1].

```
Algorithm 3.2: Ancestor_id (octant o, integer \(l\) )
    Input \(: 0<l \leq o . l\)
    Result : \(i\) such that if a. \(l=l-1\) and \(o \in \operatorname{desc}(\mathrm{a})\), then \(\circ \in \operatorname{desc}(\operatorname{child}(\mathrm{a})[i])\)
    \(\overline{h \leftarrow 2^{l_{\max }-l} \quad \text { [the }\left(l_{\max }-l\right) \text { th bits of the coordinates } 0 . x \text { describe the ancestor with level } l \text { ] }}\)
    \(i \leftarrow 0\)
    for all \(0 \leq j<d\) do
        \(i \leftarrow i \mid\left(\left(0 . x_{j} \& h\right) ? 2^{j}: 0\right) \quad[" \mid "\) and "\&" are bitwise OR and AND]
    return \(i\)
```

If we applied Ancestor_id to each octant in $\mathbf{A}$, we would get a monotonic sequence of integers, so if we search $\mathbf{A}$ with the key $i$ and use Ancestor_id to test equality, the lowest matching index will give the first descendant of child(a)[i] in $\mathbf{A}$. The split operation, however, is used repeatedly, both by Search and by the algorithm Iterate we will present in Section 5, so to make the procedure as efficient as possible, we combine these searches into one algorithm Split_array (Algorithm 3.3), which is essentially an efficient binary search for a sorted list of keys.

```
Algorithm 3.3: Split_array (octant array A, octant a)
    Input : \(\mathbf{A}\) is sorted; \(\mathbf{a}\) is a strict ancestor of \(\mathbf{A}[j]\) for each \(j, \mathbf{A} \subseteq \operatorname{desc}(\mathbf{a}) \backslash\{\mathbf{a}\}\).
    Result : \(\forall 0 \leq i<2^{d}, \mathbf{H}[i]\) is a sorted array containing \(\operatorname{desc}(\operatorname{child}(a)[i]) \cap \mathbf{A}\).
    \(\overline{\mathbf{k}[0] \leftarrow 0 \quad \text { [invariant } \mathbf{1} \forall \text { i: if } j \geq \mathbf{k}[i], \ldots \quad \text { ] }}\)
    for all \(1 \leq i \leq 2^{d}\) do \(\mathbf{k}[i] \leftarrow|\mathbf{A}| \quad[\ldots\) then Ancestor_id \((\mathbf{A}[j]\), a.l \(l+1) \geq i]\)
    for \(i=1\) to \(2^{d}-1\) do
        \(m \leftarrow \mathbf{k}[i-1] \quad\) [invariant 2: if \(j<m\), then Ancestor_id \((\mathbf{A}[j]\), a.l \(l+1)<i]\)
        while \(m<\mathbf{k}[i]\) do
            \(n \leftarrow m+\lfloor(\mathbf{k}[i]-m) / 2\rfloor \quad[\mathbf{k}[i-1] \leq m \leq n<\mathbf{k}[i]]\)
            \(c \leftarrow\) Ancestor_id \((\mathbf{A}[n]\), a. \(l+1) \quad[\mathbf{A}[n] \in \operatorname{desc}(\operatorname{child}(\mathbf{a})[c])]\)
            if \(c<i\) then
                                \([\mathbf{A}[n]\) is a descendant of a previous child]
                \(m \leftarrow n+1 \quad\) [increase lower bound to maintain invariant 2]
            else \([\mathbf{A}[n]\) is a descendant of the cth child, \(c \geq i]\)
                for all \(i \leq j \leq c\) do \(\mathbf{k}[j] \leftarrow n \quad\) [decrease \(\mathbf{k}[j]\) to maintain invariant 1]
    for all \(0 \leq i<2^{d}\) do \(\mathbf{H}[i] \leftarrow\) alias \(\mathbf{A}[\mathbf{k}[i], \ldots, \mathbf{k}[i+1]-1]\)
    return H
```

4. Constructing ghost layers for unbalanced forests. As discussed in Section 2.4 there is no a-priori knowledge on any given process about what leaves might be in a neighboring process's partition. This knowledge, however, is necessary to determine the local neighborhoods of leaves that are adjacent to inter-process boundaries, which is crucial to many application-level algorithms. If a forest of octrees obeys a 2:1 balance condition, it is known that a leaf's neighbors in other partitions can differ by at most one refinement level. The previously presented algorithm Ghost [6, Algorithm 20] uses this fact to identify neighboring processes and communicate leaves between them. Ghost is short and effective, but not usable for an unbalanced forest. Here we present an algorithm for ghost layer construction that works for all forests. Its key component is a recursive algorithm that determines when a leaf's domain and a process's subdomain are adjacent to each other.
4.1. Ghost layer construction using intersection tests. A leaf o $\notin \mathcal{O}_{\mathfrak{q}}$ is in the full ghost layer for process $\mathfrak{q}$ if its boundary intersects the subdomain's closure, $\partial \operatorname{Dom}(o) \cap \overline{\Omega_{\mathfrak{q}}} \neq \emptyset$. This definition includes leaves whose intersection with $\overline{\Omega_{\mathfrak{q}}}$ is a single vertex. Some applications, such as discontinuous Galerkin finite element methods, only require a ghost layer to include leaves whose intersections with $\overline{\Omega_{\mathfrak{q}}}$ have codimension 1. The boundary set bound(o) 2.10 allows us to define a ghost layer parametrized by codimension. For $1 \leq k \leq d$, the $\mathfrak{p}$-to- $\mathfrak{q}$ ghost layer $\mathbf{G}_{\mathfrak{p} \rightarrow \mathfrak{q}}^{k}$ is a sorted array containing the subset of the leaves $\mathcal{O}_{\mathfrak{p}}$ whose boundaries intersect $\mathfrak{q}$ 's subdomain at a point with codimension less than or equal to $k$,

$$
\begin{equation*}
\mathbf{G}_{\mathfrak{p} \rightarrow \mathfrak{q}}^{k}:=\left\{o \in \mathcal{O}_{\mathfrak{p}}: \exists \mathrm{c} \in \operatorname{bound}(\mathrm{o}), \operatorname{dim}(\mathrm{c}) \geq d-k, \operatorname{Dom}(\mathrm{c}) \cap \overline{\Omega_{\mathfrak{q}}} \neq \emptyset\right\} \tag{4.1}
\end{equation*}
$$

The $k$-ghost layer $\mathbf{G}_{\mathfrak{p}}^{k}$ for process $\mathfrak{p}$ is the sorted union of all $\mathfrak{q}$-to- $\mathfrak{p}$ ghost layers,

$$
\begin{equation*}
\mathbf{G}_{\mathfrak{p}}^{k}:=\bigsqcup_{0 \leq \mathfrak{q}<P, \mathfrak{q} \neq \mathfrak{p}} \mathbf{G}_{\mathfrak{q} \rightarrow \mathfrak{p}}^{k} \tag{4.2}
\end{equation*}
$$

To construct all $\mathbf{G}_{\mathfrak{p} \rightarrow \mathfrak{q}}^{k}$ according to (4.1), process $\mathfrak{p}$ would have to perform intersection tests between the boundary set of every leaf in $\mathcal{O}_{\mathfrak{p}}$ and every other process's subdomain. We can reduce the number of intersection tests by noting that if $\operatorname{Dom}(\mathrm{c}) \cap \overline{\Omega_{\mathfrak{q}}} \neq \emptyset$, then by a corollary to proposition $2.6, \overline{\Omega_{\mathfrak{q}}}$ must overlap some octant s in the support set $\operatorname{supp}(\mathrm{c})$ that surrounds c. Locating the first and last process subdomains that overlap $\operatorname{Dom}(\mathbf{s})$ for $\mathrm{s} \in \operatorname{supp}(\mathrm{c})$ takes $O(\log P)$ time and typically reduces the number of intersection tests per point from $O(P)$ to $O(1)$. We give pseudocode for computing which $\mathfrak{p}$-to- $\mathfrak{q}$ ghost layers contain a leaf o in Algorithm4.1.

```
Algorithm 4.1: Add_ghost (octant o, integer \(k\) )
    Input : leaf octant \(\circ \in \mathcal{O}_{\mathfrak{p}} ; 1 \leq k \leq d\).
    Result : the set \(\mathcal{Q}\) of processes \(\mathfrak{q}\) such that \(\circ \in \mathbf{G}_{\mathfrak{p} \rightarrow \mathfrak{q}}^{k}\) 4.1.
    \(\overline{\mathcal{Q} \leftarrow \emptyset}\)
    for all \(\mathrm{c} \in \operatorname{bound}(\mathrm{o})\) such that \(\operatorname{dim}(\mathrm{c}) \geq d-k\) do
        for all \(s \in \operatorname{supp}(c) \backslash\{0\}\) do
            \(\left(\mathrm{f}_{\mathrm{s}}, \mathrm{l}_{\mathrm{s}}\right) \leftarrow \operatorname{range}(\mathrm{s})\)
            \(\left(\mathfrak{q}_{\text {first }}, \mathfrak{q}_{\text {last }}\right) \leftarrow\left(\operatorname{locate}\left(\mathbf{f}_{s}\right)\right.\), locate \(\left.\left(\mathbf{l}_{s}\right)\right) \quad\) [overlapping processes]
            for all \(\mathfrak{q}_{\text {first }} \leq \mathfrak{q} \leq \mathfrak{q}_{\text {last }}, \mathfrak{q} \neq \mathfrak{p}\) do
                \(\left(\mathrm{f}_{\mathfrak{q}}, \mathrm{l}_{\mathfrak{q}}\right) \leftarrow \operatorname{range}(\mathfrak{q}) \quad\left[\overline{\Omega_{\mathfrak{q}}}=\overline{\bigcup \operatorname{Dom}\left(\left[\mathrm{f}_{\mathfrak{q}}, 1_{\mathfrak{q}}\right]\right)}\right]\)
                \((f, l) \leftarrow\left(\max \left\{\mathrm{f}_{\mathbf{s}}, \mathrm{f}_{\mathfrak{q}}\right\}, \min \left\{\mathbf{1}_{\mathbf{s}}, \mathrm{l}_{\mathfrak{q}}\right\}\right) \quad\left[\overline{\mathrm{UDom}([\mathrm{f}, \mathrm{l}])}=\overline{\Omega_{\mathfrak{q}}} \cap \overline{\operatorname{Dom}(\mathbf{s})}\right]\)
                if \(\overline{\bigcup \operatorname{Dom}([f, 1])} \cap \operatorname{Dom}(c) \neq \emptyset\) then \(\mathcal{Q} \leftarrow \mathcal{Q} \cup\{\mathfrak{q}\}\)
    return \(\mathcal{Q}\)
```



| $\mathcal{B}_{\cap}^{0}$ | $\left\{c_{0}, e_{0}, e_{4}, e_{8}, f_{0}, f_{2}, f_{4}\right\}$ |
| :--- | :--- |
| $\mathcal{B}_{\cap}^{1}$ | $\left\{c_{1}, e_{0}, e_{5}, e_{9}, f_{1}, f_{2}, f_{4}\right\}$ |
| $\mathcal{B}_{\cap}^{2}$ | $\left\{c_{2}, e_{1}, e_{4}, e_{10}, f_{0}, f_{3}, f_{4}\right\}$ |
| $\mathcal{B}_{\cap}^{3}$ | $\left\{c_{3}, e_{1}, e_{5}, e_{11}, f_{1}, f_{3}, f_{4}\right\}$ |
| $\mathcal{B}_{\cap}^{4}$ | $\left\{c_{4}, e_{2}, e_{6}, e_{8}, f_{0}, f_{2}, f_{5}\right\}$ |
| $\mathcal{B}_{\cap}^{5}$ | $\left\{c_{5}, e_{2}, e_{7}, e_{9}, f_{1}, f_{2}, f_{5}\right\}$ |
| $\mathcal{B}_{\cap}^{6}$ | $\left\{c_{6}, e_{3}, e_{6}, e_{10}, f_{0}, f_{3}, f_{5}\right\}$ |
| $\mathcal{B}_{\cap}^{7}$ | $\left\{c_{7}, e_{3}, e_{7}, e_{11}, f_{1}, f_{3}, f_{5}\right\}$ |

Fig. 4.1. (left) An illustration of a specific instance of proposition 4.1. If the range [f, 1] contains only descendants of child(s)[2], then its domain intersects $\operatorname{Dom}((\mathbf{s}, b))$ (left) if and only if it intersects $\operatorname{Dom}((\operatorname{child}(\mathrm{s})[i], b))$ (right) and $b \in \mathcal{B}_{\cap}^{2}$. (right) The child-boundary intersections are enumerated.
4.2. Finding a range's boundaries recursively. The kernel of Algorithm 4.1 is the intersection test on line 9 ,

$$
\begin{equation*}
(\overline{\bigcup \operatorname{Dom}([f, l])} \cap \operatorname{Dom}(c)=\emptyset ?) \tag{4.3}
\end{equation*}
$$

where $c$ is a point, $[f, l]$ is the range between two atoms, and $f$ and $l$ are both descendants of an octant $s$ in the support set $\operatorname{supp}(c)$. We must specify how this intersection test is to be performed. Because $\mathbf{s} \in \operatorname{supp}(\mathrm{c})$ implies $\mathrm{c} \in \operatorname{bound}(\mathbf{s})$ (see propositions 2.7 and 2.8 , the point c must be equal to ( $\mathrm{s}, b$ ) for some boundary index $b \in \mathcal{B}$, so we can rephrase the test as $(\overline{\bigcup \operatorname{Dom}([\mathbf{f}, \mathbf{1}])} \cap \operatorname{Dom}((\mathbf{s}, b))=\emptyset$ ? ). This test is a specific case of the problem of constructing the range-boundary intersection $\mathcal{B}_{\cap}(\mathrm{f}, \mathrm{l}, \mathrm{s})$, the set of all boundary indices $b$ such that $\operatorname{Dom}((\mathrm{s}, b))$ intersects the range,

$$
\begin{equation*}
\mathcal{B}_{\cap}(\mathrm{f}, 1, \mathrm{~s}):=\{b \in \mathcal{B}: \overline{\bigcup \operatorname{Dom}([\mathrm{f}, \mathrm{l}])} \cap \operatorname{Dom}((\mathbf{s}, b)) \neq \emptyset\} \tag{4.4}
\end{equation*}
$$

If the range $[\mathrm{f}, \mathrm{l}]$ contains only descendants of some child of $\mathbf{s}$, say child( $(\mathbf{s})[i]$, then the range-boundary intersection must be a subset of the child-boundary intersection $\mathcal{B}_{\cap}^{i}$, which is the subset of $\mathcal{B}$ corresponding to points in bound(s) intersected by $\overline{\text { Dom(child(s) }[i])}$,

$$
\begin{equation*}
\mathcal{B}_{\cap}^{i}:=\{b \in \mathcal{B}: \overline{\operatorname{Dom}(\operatorname{child}(\mathbf{s})[i])} \cap \operatorname{Dom}((\mathbf{s}, b)) \neq \emptyset\} . \tag{4.5}
\end{equation*}
$$

The child-boundary intersections $\left\{\mathcal{B}_{\cap}^{i}\right\}_{0 \leq i<2^{d}}$ (Figure 4.1. right) are the same for all (non-atom) octants. The following proposition, illustrated in Figure 4.1 (left), shows how the child-boundary intersection $\mathcal{B}_{\cap}^{i}$ relates $\mathcal{B}_{\cap}(\mathbf{f}, 1, \operatorname{child}(\mathbf{s})[i])$ to $\mathcal{B}_{\cap}(\mathbf{f}, 1, \mathbf{s})$.

Proposition 4.1 (Range-boundary intersection recursion). If $f$ and 1 are atoms, $\mathrm{f} \leq 1$, and both are descendants of child(s) $[i]$, then

$$
\begin{equation*}
\left(b \in \mathcal{B}_{\cap}(\mathbf{f}, 1, \mathbf{s})\right) \Leftrightarrow\left(b \in \mathcal{B}_{\cap}(\mathbf{f}, 1, \operatorname{child}(\mathbf{s})[i]) \cap \mathcal{B}_{\cap}^{i}\right) . \tag{4.6}
\end{equation*}
$$

This result allows us to construct $\mathcal{B}_{\cap}(f, 1, s)$ by partitioning $[\mathbf{f}, 1]$ into ranges for all of the overlapping children,

$$
\begin{equation*}
[\mathrm{f}, \mathrm{l}]=\bigsqcup_{i \in \mathcal{I}}\left[\mathrm{f}_{i}, \mathrm{l}_{i}\right], \quad \mathcal{I}:=\{i: \exists \mathrm{a} \in \operatorname{desc}(\operatorname{child}(\mathbf{s})[i]), \mathrm{f} \leq \mathrm{a} \leq \mathrm{l}\}, \tag{4.7}
\end{equation*}
$$

and constructing the range-boundary intersection for those children,

$$
\begin{equation*}
\mathcal{B}_{\cap}(\mathrm{f}, \mathrm{l}, \mathrm{~s})=\bigcup_{i \in \mathcal{I}} \mathcal{B}_{\cap}\left(\mathbf{f}_{i}, 1_{i}, \operatorname{child}(\mathrm{~s})[i]\right) \cap \mathcal{B}_{\cap}^{i} \tag{4.8}
\end{equation*}
$$

This leads to the recursive algorithm Find_range_boundaries (Algorithm4.2), which computes $\mathcal{B}_{\cap}(f, l, s) \cap \mathcal{B}_{\text {query }}$ for a set $\mathcal{B}_{\text {query }} \subseteq \mathcal{B}$.

```
Algorithm 4.2: Find_range_boundaries (octants f, 1 , s, index set \(\mathcal{B}_{\text {query }}\) )
    Input : f and l are atom descendants of \(\mathrm{s}, \mathrm{f} \leq 1 ; \mathcal{B}_{\text {query }} \subseteq \mathcal{B}\).
    Result : \(\mathcal{B}_{\cap}(\mathrm{f}, \mathrm{l}, \mathrm{s}) \cap \mathcal{B}_{\text {query }}\) 4.4).
    if \(\mathcal{B}_{\text {query }}=\emptyset\) or s.l \(=l_{\text {max }}\) then return \(\mathcal{B}_{\text {query }}\)
    \(j \leftarrow\) Ancestor_id ( \(\mathbf{f}, \mathrm{s} . l+1) \quad\) [index of first child whose range overlaps \([\mathrm{f}, \mathrm{l}]\) ]
    \(k \leftarrow\) Ancestor_id ( \(1, \mathrm{~s} . l+1\) ) [index of last child whose range overlaps \([\mathrm{f}, \mathrm{l}]\) ]
    if \(j=k\) then return Find_range_boundaries ( \(\left.\mathbf{f}, 1, \operatorname{child}(\mathbf{s})[j], \mathcal{B}_{\text {query }} \cap \mathcal{B}_{\cap}^{j}\right)\)
    \(\mathcal{B}_{\text {match }} \leftarrow \bigcup_{j<i<k} \mathcal{B}_{\text {query }} \cap \mathcal{B}_{\cap}^{i} \quad\) [boundary touched by wholly-covered children]
    \(\mathcal{B}_{\text {match }}^{j} \leftarrow\left(\mathcal{B}_{\text {query }} \cap \mathcal{B}_{\cap}^{j}\right) \backslash \mathcal{B}_{\text {match }}\)
    \(\left(\mathbf{f}_{j}, 1_{j}\right) \leftarrow \operatorname{range}(\operatorname{child}(\mathbf{s})[j]) \quad\left[[\mathbf{f}, \mathbf{l}] \cap\left[\mathbf{f}_{j}, 1_{j}\right]=\left[\mathbf{f}, 1_{j}\right]:\right.\) if \(\mathbf{f} \neq \mathbf{f}_{j}\), recursion is needed \(]\)
    if \(\mathrm{f} \neq \mathrm{f}_{j}\) then \(\mathcal{B}_{\text {match }}^{j} \leftarrow\) Find_range_boundaries \(\left(\mathrm{f}, 1_{j}\right.\), child \(\left.(\mathbf{s})[j], \mathcal{B}_{\text {match }}^{j}\right)\)
    \(\mathcal{B}_{\text {match }}^{k} \leftarrow\left(\mathcal{B}_{\text {query }} \cap \mathcal{B}_{\cap}^{j}\right) \backslash \mathcal{B}_{\text {match }} \backslash B_{\text {match }}^{j}\)
    \(\left(\mathrm{f}_{k}, 1_{k}\right) \leftarrow \operatorname{range}(\operatorname{child}(\mathbf{s})[k]) \quad\left[[\mathrm{f}, 1] \cap\left[\mathrm{f}_{k}, 1_{k}\right]=\left[\mathrm{f}_{k}, \mathbf{1}\right]:\right.\) if \(\mathrm{l} \neq \mathrm{l}_{k}\), recursion is needed \(]\)
    if \(1 \neq 1_{k}\) then \(\mathcal{B}_{\text {match }}^{k} \leftarrow\) Find_range_boundaries \(\left(f_{k}, 1\right.\), child \(\left.(\mathbf{s})[k], \mathcal{B}_{\text {match }}^{k}\right)\)
    return \(\mathcal{B}_{\text {match }} \cup \mathcal{B}_{\text {match }}^{j} \cup \mathcal{B}_{\text {match }}^{k}\)
```

To compute the intersection test $(\overline{\bigcup \operatorname{Dom}([f, l])} \cap \operatorname{Dom}((\mathbf{s}, b))=\emptyset$ ?) in Algorithm 4.1, we choose $\mathcal{B}_{\text {query }}=\{b\}$ and use Find_range_boundaries to check whether $\left(\mathcal{B}_{\cap}(\mathrm{f}, 1, \mathrm{~s}) \cap \mathcal{B}_{\text {query }}=\emptyset\right.$ ? ). A proof of the correctness of Algorithm 4.2 is given in Appendix A The recursive procedure is also illustrated in Figure 4.2
4.3. Notes on implementation. Ghost layer construction in Ghost has a few optimizations not given in the pseudocode in Algorithm 4.1. Most leaves in $\mathcal{O}_{\mathfrak{p}}$ do not touch the boundary of $\overline{\Omega_{\mathfrak{p}}}$, and so cannot be in $\mathbf{G}_{\mathfrak{p} \rightarrow \mathfrak{q}}^{k}$ for any $\mathfrak{q} \neq \mathfrak{p}$. To avoid the intersection tests for these leaves, we first check to see if o's $3 \times 3$ neighborhood (or "insulation layer" 23 ) is owned by $\mathfrak{p}$ : this can be accomplished with two comparisons, for the first and last atoms of the neighborhood, against the first-atoms array $\mathbf{f} 2.21$. We also note that if c is a 0 -point then $\overline{\Omega_{\mathfrak{q}}}$ intersects $\operatorname{Dom}(\mathrm{c})$ if and only if $\mathfrak{q}=$ locate( $\left.\mathbf{s}\right)$ for some atom s in c's atomic support set 2.12 . Because this simpler test is available, we do not call Find_range_boundaries for 0-points.

If an instance of Find_range_boundaries calls two recursive copies of itself, all future instances will call only one recursive copy. We use this fact to take advantage of tail-recursion optimization in our implementation. Because $|\mathcal{B}|<32$ for $d=2$ and $d=3$, we can perform the set intersection, unions, and differences in Algorithm 4.2 by assigning each $b \in \mathcal{B}$ to a bit in an integer and performing bitwise operations.
5. A universal topology iterator. A forest is first of all a storage scheme for a mesh refinement topology. Applications use this topological information in ways that we do not wish to restrict or anticipate. We focus instead on designing an interface that conveys this information to applications in a complete and efficient manner, with the main goal of minimizing the points of contact between p4est on the one hand and the application on the other.


FIG. 4.2. An illustration of Find_range_boundaries, listed in Algorithm 4.2. (top left) Red indicates the points in bound(s) indexed by $\mathcal{B}_{\text {query }}$; the children containing f and 1 are determined. (top right) The contribution to $\mathcal{B} \cap(\mathbf{f}, \mathrm{l}, \mathrm{s}) \cap \mathcal{B}_{\text {query }}$ (green) of children in the middle of the range is calculated; light blue indicates that this portion of the range $[\mathrm{f}, \mathrm{l}]$ has been processed. (bottom left) The arguments for the recursive calls are constructed. (bottom right) The sets returned by the recursive calls are added to the return set.

As we will see in our discussion of a specific node numbering algorithm in Section 6 some applications need to perform operations not just on leaves, but also on their boundary points. Our algorithm that facilitates this is called Iterate.

Remark 5.1. The algorithm Iterate requires that the leaves in the micro layer $\mathcal{O}$ are 2:1 balanced. This is assumed for the remainder of this section and Section 6
5.1. Definitions. The algorithm Iterate is distributed and communicationfree (assuming that the full ghost layer $\mathbf{G}_{\mathfrak{p}}^{d}(4.2$ has already been constructed): on process $\mathfrak{p}$, it takes a user-supplied callback and executes it once for every leaf and leaf-boundary point c that is relevant to $\mathcal{O}_{\mathfrak{p}}$, supplying information about the neighborhood around c. We define exactly what this means here.

The union of all leaves with their closure sets, $\bigcup \operatorname{clos}(\mathcal{O})$, defines a covering of $\Omega$, $\bar{\Omega}=\bigcup \operatorname{Dom}(\bigcup \operatorname{clos}(\mathcal{O}))$. This covering may not be a partition, because $\bigcup \operatorname{clos}(\mathcal{O})$ may contain hanging points: $n$-points of dimension $n<d$ that are in the child partition sets 2.7 ) of other points in $\bigcup \operatorname{clos}(\mathcal{O})$. We can define a partition by removing these hanging points. The global partition $\mathcal{P}_{\Omega}$ is

$$
\begin{equation*}
\mathcal{P}_{\Omega}:=\bigcup \operatorname{clos}(\mathcal{O}) \backslash\{\mathrm{c}: \exists \mathrm{e} \in \bigcup \operatorname{clos}(\mathcal{O}), \mathrm{c} \in \operatorname{part}(\mathrm{e})\} \tag{5.1}
\end{equation*}
$$

If there is just one process, the function Iterate executes a user-supplied callback function for every point $c \in \mathcal{P}_{\Omega}$. For a distributed forest, Iterate as called by process $\mathfrak{p}$ executes the callback function only for the subset of $\mathcal{P}_{\Omega}$ that is relevant to $\mathcal{O}_{\mathfrak{p}}$. In


FIG. 5.1. Suppose process $\mathfrak{p}$ owns only octant $\circ$ in this two-dimensional illustration. $\mathcal{P}_{\Omega}$ is the set of all points shown: note that because some points in clos(o) are hanging, they are not included. The set $\mathcal{P}_{\mathfrak{p}}$ of points that overlap $\overline{\Omega_{p}}$ is shown in blue. The 0-point c shown in red is not in $\mathcal{P}_{\mathfrak{p}}$, but is in its closure, $\overline{\mathcal{P}_{\mathfrak{p}}}$.
the p4est implementation, we allow for two definitions of what is relevant. The first is the locally relevant partition, which is the subset $\mathcal{P}_{\mathfrak{p}} \subseteq \mathcal{P}_{\Omega}$ that overlaps $\overline{\Omega_{\mathfrak{p}}}$,

$$
\begin{equation*}
\mathcal{P}_{\mathfrak{p}}:=\left\{c \in \mathcal{P}_{\Omega}: \exists o \in \mathcal{O}_{\mathfrak{p}}, \operatorname{Dom}(\mathrm{c}) \cap \overline{\operatorname{Dom}(o)} \neq \emptyset\right\} . \tag{5.2}
\end{equation*}
$$

One potential problem with $\mathcal{P}_{\mathfrak{p}}$ is that, because of hanging points, it may not be closed: if $c \in \mathcal{P}_{\mathfrak{p}}$, there may be $\mathrm{e} \in \operatorname{clos}(\mathrm{c})$ such that $\mathrm{e} \notin \mathcal{P}_{\mathfrak{p}}$. As we will show in Section 6, closedness is necessary for some applications, so we also define the closed locally relevant partition $\overline{\mathcal{P}_{\mathfrak{p}}}$,

$$
\begin{equation*}
\overline{\mathcal{P}_{\mathfrak{p}}}:=\bigcup\left\{\operatorname{clos}(\mathrm{e}): \mathrm{e} \in \mathcal{P}_{\mathfrak{p}}\right\} . \tag{5.3}
\end{equation*}
$$

The sets we have defined thus far- $\mathcal{P}_{\Omega}, \mathcal{P}_{\mathfrak{p}}$, and $\overline{\mathcal{P}_{\mathfrak{p}}}$-are illustrated in Figure 5.1 .
If Iterate only supplied the callback with each relevant point $c$ in isolation, its utility would be limited, because it would say nothing about the neighborhood around c. We call the neighborhood of adjacent leaves the leaf support set leaf $\operatorname{supp}(\mathrm{c})$,

$$
\begin{equation*}
\text { leaf } \operatorname{supp}(c):=\{o \in \mathcal{O}: \overline{\operatorname{Dom}(o)} \cap \operatorname{Dom}(c) \neq \emptyset\} . \tag{5.4}
\end{equation*}
$$

Note that the leaf support set leaf $\operatorname{supp}(c)$ may differ from the support set supp(c) (2.11), which is independent of the leaves in the micro layer $\mathcal{O}$. Because process $\mathfrak{p}$ does not have access to all leaves in $\mathcal{O}$, it can only compute the subset of the leaf support set that is contained in the local leaves and in the full ghost layer $\mathbf{G}_{\mathfrak{p}}^{d} 4.2$, which we call the local leaf support set leaf $\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})$,

$$
\begin{equation*}
\text { leaf } \operatorname{supp}_{\mathfrak{p}}(\mathrm{c}):=\text { leaf } \operatorname{supp}(\mathrm{c}) \cap\left(\mathcal{O}_{\mathfrak{p}} \cup \mathbf{G}_{\mathfrak{p}}^{d}\right) \tag{5.5}
\end{equation*}
$$

Proposition 5.2. leaf supp $p_{\mathfrak{p}}(\mathrm{c})=$ leaf $\operatorname{supp}(\mathrm{c})$ if and only if $\mathrm{c} \in \mathcal{P}_{\mathfrak{p}}$.
The local leaf support set, though it may not contain all of the leaf support set, is what Iterate supplies to the user-supplied callback to describe the neighborhood around c. The local leaf support set can be used to determine if $\mathrm{c} \in \mathcal{P}_{\mathfrak{p}}$ or $\mathrm{c} \in \overline{\mathcal{P}_{\mathfrak{p}}}$, using a function Is_relevant (Algorithm 5.1).

```
Algorithm 5.1: Is_relevant (point c, octant set leaf supp \(p_{p}(\mathrm{c})\) )
    Input : leaf \(\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})\) is the local leaf support set of c 5.5 .
    Result : true if and only if \(c\) is in the relevant set ( \(\mathcal{P}_{\mathfrak{p}}\) or \(\overline{\mathcal{P}_{\mathfrak{p}}}\) ).
    for all \(\mathrm{s} \in \operatorname{leaf} \operatorname{supp}_{\mathfrak{p}}(\mathrm{c})\) do
        if \(s \in \mathcal{O}_{\mathfrak{p}}\) then return true
        else
            for all \(\mathrm{e} \in\) bound(s) such that \(\mathrm{c} \in\) bound(e) do
                if \(\operatorname{Dom}(\mathrm{e}) \cap \overline{\Omega_{\mathfrak{p}}} \neq \emptyset\) then return true [intersection test from Section 4.2]
    return false
```

For each octant $o \in \operatorname{leaf} \operatorname{supp}_{\mathfrak{p}}(c)$, the p4est implementation of Iterate supplements the octant data fields o.l, o. $x$, and o.t with additional data passed to the callback function. We supply a boolean identifying whether $o \in \mathcal{O}_{\mathfrak{p}}$, so no searching is necessary to determine if o is local or a ghost. We also supply the index of o within either $\mathbf{O}_{\mathfrak{p}}^{t}$ for $t=$ o.t (which is easily converted to $j$ such that $\mathcal{O}_{\mathfrak{p}}[j]=0$ ) or within the ghost layer $\mathbf{G}_{\mathfrak{p}}^{d} 4.2$. Keeping track of this information does not change the computational complexity of Iterate, but introduces additional bookkeeping that we will omit from our presentation of the algorithm.
5.2. Iterating in the interior of a point. A simple implementation of Iterate might take each leaf $o \in \mathcal{O}_{\mathfrak{p}}$ in turn and, for each $c \in \operatorname{bound}(o)$, compute leaf $\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})$ by searching through $\mathcal{O}_{\mathfrak{p}}$ and $\mathbf{G}_{\mathfrak{p}}^{d}$ to find o's neighbors that are adjacent to $c$. A bounded number of binary searches would be performed per leaf, so the total iteration time would be $O\left(N_{\mathfrak{p}} \log N_{\mathfrak{p}}\right)$. This is the strategy used by the Nodes algorithm in p4est [6, Algorithm 21] and by other octree libraries [26]. We note two problems with this approach. The first is the large number of independent searches that are performed. The second is that this approach needs some way of ensuring that the callback is executed for each relevant point only once, such as storing the set of points for which the callback has executed in a hash table.

Instead, the implementation of Iterate that we present proceeds recursively. We take as inputs to the recursive procedure a point $c$ and a set of arrays $\mathbf{S}$, where $\mathbf{S}[i]$ contains all leaves that are descendants of the support set octant $\operatorname{supp}(c)[i]$. If $c$ is in the global partition $\mathcal{P}_{\Omega}$ and is in the relevant set $\left(\mathcal{P}_{\mathfrak{p}}\right.$ or $\left.\overline{\mathcal{P}_{\mathfrak{p}}}\right)$, then the leaves in leaf $\operatorname{supp}_{\mathfrak{p}}(c)$ can be found in the $\mathbf{S}$ arrays and the callback function can be executed. Otherwise, the points of the global partition $\mathcal{P}_{\Omega}$ that are descendants of c can be divided between the points in the child partition set part(c) 2.7. Each e $\in \operatorname{part}(\mathrm{c})$ takes the place of $c$ in a call to the recursive procedure: to compute the new $\mathbf{S}$ arrays for e, we use the function Split_array (described in Section 3.2) on the original arrays $\mathbf{S}$. This is spelled out in Iterate_interior (Algorithm 5.2].

We provide some figures to illustrate the recursion in Iterate_interior: Figure 5.2 shows the cases when $\operatorname{dim}(c)=d$ and $0<\operatorname{dim}(c)<d$ and Figure 5.3 shows the case when $\operatorname{dim}(c)=0$. The correctness of Algorithm 5.2 is proved in Appendix B

REMARK 5.3. An instance of Iterate_interior may call multiple recursive copies of itself: one for each point in the child partition set part(c) (see the loop starting on line 22. We have not yet specified an order for these recursive calls. In our implementation, we have chosen to order these calls by decreasing point dimension. This guarantees that, if $\mathrm{e} \in$ bound $(\mathrm{c})$, then the callback is executed for c before it is executed for e. We take advantage of this order in designing a node-numbering algorithm in Section 6
5.3. Iterating on a forest. To iterate on the complete forest, each process must call Iterate_interior for the closure set of the root of every tree. This is listed in Iterate (Algorithm 5.3). Asymptotic analysis of the performance of Iterate is given in Appendix Chows that, in general, Iterate executes in $\mathcal{O}\left(\left(\left(\max _{0 \in \mathcal{O}}\right.\right.\right.$ o.l $)+$ $\left.\left.N_{\mathfrak{p}}\right) \log N_{\mathfrak{p}}\right)$ time, but if the refinement pattern of the octrees in the forest is uniform or nearly so, then it executes in $\mathcal{O}\left(\log P+N_{\mathfrak{p}}\right)$ time.
5.4. Implementation. The implementation of Iterate in p4est has some differences from the presented algorithm to optimize performance. Iterate_interior is implemented in while-loop form to keep the stack from growing: all space needed for recursion (which is $O\left(l_{\max }\right)$ ) is pre-allocated on the heap. We noticed in early




Fig. 5.2. Illustrations of Iterate_interior for $\operatorname{dim}(c)=d$ (top) and $0<\operatorname{dim}(c)<d$ (bottom). The color red indicates the argument point c of Iterate_interior. The dotted squares indicate the arrays $\mathbf{S}[i]$ of leaves in $\mathcal{O}$ that are descendants of $\operatorname{supp}(c)[i]$. If $\operatorname{supp}(c)[i]$ is a leaf (which is determined by testing whether $\mathbf{S}[i]=\{\operatorname{supp}(c)[i]\})$, then it is in leaf $\operatorname{supp}(c)$, which is shown with the color blue; otherwise, $\mathbf{S}[i]$ is split using Split_array. If a leaf has been found, the user-supplied callback function is executed, which we indicate with the color green; otherwise, Iterate_interior is called for each point in the child partition set part(c): the support sets for these points are found in the children of the octants in $\operatorname{supp}(\mathrm{c})$, and the arrays of leaves contained in them are found in the sets created by Split_array.

```
Algorithm 5.2: Iterate_interior (point c, octant arrays S, callback)
    Input : \(\mathbf{S}[i]\) is the sorted array of all leaves \(o \in \mathcal{O}_{p} \cup \mathbf{G}_{\mathfrak{p}}^{d}\) such that \(o \in \operatorname{desc}(\operatorname{supp}(c)[i])\).
    Result : if \(\mathrm{c} \in \mathcal{P}_{\mathfrak{p}}\) (or \(\overline{\mathcal{P}_{\mathfrak{p}}}\) ), then leaf \(\operatorname{supp}_{\mathfrak{p}}(\mathrm{c}) \sqrt{5.5}\) ) is computed and passed to callback;
            otherwise, callback is called for each \(\mathbf{e} \in \mathcal{P}_{\mathfrak{p}}\) (or \(\overline{\mathcal{P}_{\mathfrak{p}}}\) ) such that \(\operatorname{Dom}(\mathrm{e}) \subset \operatorname{Dom}(\mathrm{c})\).
    \(\overline{\text { if } \cup \mathbf{S} \cap \mathcal{O}_{\mathfrak{p}}=\emptyset \text { then return [stop if there are no local leaves] }}\)
    octant set \(\mathcal{L} \leftarrow \emptyset \quad \quad\) if \(\mathrm{c} \in \mathcal{P}_{\Omega}\), then \(\mathcal{L}\) will equal leaf \(\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})\) ]
    boolean stop \(\leftarrow\) false
                            [stop will be true if \(\mathrm{c} \in \mathcal{P}_{\Omega}\) ]
[see Figure 5.2]
    if \(\operatorname{dim}(c)>0\) then
                            [stop will be true if \(\mathrm{c} \in \mathcal{P}_{\Omega}\) ]
[see Figure 5.2 ]
        for all \(0 \leq i<|\operatorname{supp}(\mathrm{c})|\) do
            \(\mathrm{s} \leftarrow \operatorname{supp}(\mathrm{c})[i]\)
            if \(\mathbf{S}[i]=\{\mathbf{s}\}\) then
                    stop \(\leftarrow\) true
                \(\mathcal{L} \leftarrow \mathcal{L} \cup\{\mathbf{s}\}\)
                            [then \(\mathrm{c} \in \mathcal{P}_{\Omega}, \ldots\) ]
                            [and \(\mathbf{s} \in\) leaf \(\operatorname{supp}(\mathrm{c})\) ]
                else
                    \(\mathbf{H}_{i} \leftarrow\) Split_array \((\mathbf{S}[i], \mathbf{s})\)
                \(\mathbf{h}_{i} \leftarrow \operatorname{child}(\mathrm{~s})\)
                \(b \leftarrow b\) such that \(\mathrm{c}=(\mathbf{s}, b)\)
                    \(\mathcal{L} \leftarrow \mathcal{L} \cup\left\{\mathbf{h}_{i}[j]: b \in \mathcal{B}_{\cap}^{j}\right\}\)
                    \([O(\log |\mathbf{S}[i]|)\) (see Section 3.2)]
                If \(\mathrm{c} \in \mathcal{P}_{\Omega}\), then by the \(2: 1\) condition, \(\ldots\) ]
                            [children next to \((\mathbf{s}, b)=\mathrm{c}\) are in leaf \(\operatorname{supp}(\mathrm{c})]\)
\(\left[\mathcal{B}_{\cap}^{j}\right.\) are child-boundary intersection sets 4.5]]
        lse
                                    [a 0-point, see Figure 5.3]
        stop \(\leftarrow\) true
                    anytime we find a 0-point between leaves, it is in \(\mathcal{P}_{\Omega}\).
        for \(0 \leq i<|\operatorname{supp}(c)|\) do [find leaves surrounding c: use proposition 2.8]
            \(\mathcal{L} \leftarrow \mathcal{L} \cup\{o \in \mathbf{S}[i]: \operatorname{atom} \operatorname{supp}(\mathrm{c})[i] \in \operatorname{desc}(\mathrm{o})\} \quad\) [requires \(O(\log |\mathbf{S}[i]|)\) search]
    if stop then
        if Is_relevant \((c, \mathcal{L})\) then callback \((c, \mathcal{L})\)
    else
        for all \(\mathrm{e} \in \operatorname{part}(\mathrm{c})\) do \(\quad\) set up recursion for each point in child partition set 2.7] ]
            for all \(0 \leq i<|\operatorname{supp}(\mathrm{e})|\) do [find descendants of support set octants]
                    \(\mathbf{S}_{\mathbf{e}}[i] \leftarrow \mathbf{H}_{j}[k]\) such that \(\mathbf{h}_{j}[k]=\operatorname{supp}(\mathrm{e})[i] \quad\) [subarrays created on line 11]
        Iterate_interior(e, \(\mathbf{S}_{\mathrm{e}}\), callback)
```

| $\mathbf{S}[2]$ | $\mathbf{S}[3]$ |
| :---: | :---: |
| $\operatorname{supp}(\mathrm{c})[2]$ | $\operatorname{supp}(\mathrm{c})[3]$ |
|  | c |
|  | O |
| $\operatorname{supp}(\mathrm{c})[0]$ |  |
|  | $\operatorname{supp}(\mathrm{c})[1]$ |
| $\mathbf{S}[0]$ | $\mathbf{S}[1]$ |


| $\mathbf{S}[2]$ | $\mathbf{S}[3]$ |
| :---: | :--- |
| $\operatorname{atom}$ | atom |
| $\operatorname{supp}(c)[2]$ | $\operatorname{supp}(c)[3]$ |
| atom | atom |
| $\operatorname{supp}(c)[0]$ | $\operatorname{supp}(c)[1]$ |
| $\mathbf{S}[0]$ | $\mathbf{S}[1]$ |



FIG. 5.3. An illustration of Iterate_interior when $\operatorname{dim}(c)=0$, using the same color conventions as Figure 5.2. The arguments of Iterate_interior are in the left panel. The yellow squares (middle panel) indicate octants in atom supp(c): these must be descendants of the leaves in leaf $\operatorname{supp}(c)$, so we use atom $\operatorname{supp}(c)[i]$ as a key to search for leaf $\operatorname{supp}(c)[i]$ in $\mathbf{S}[i]$. Once leaf $\operatorname{supp}(c)$ is found, the callback is executed (right panel).
tests that Split_array can be called with the same arguments multiple times during a call to Iterate. To avoid some of this recomputation, we keep an $O\left(l_{\max }\right)$ fixed-size cache of the array aliases produced by Split_array. We allow the user to specify a separate callback function for each dimension, so that extra recursion can be avoided. If, for example, the callback only needs to be executed for faces, then an instance of

```
Algorithm 5.3: Iterate (callback, octant array \(\mathbf{G}_{\mathfrak{p}}^{d}\) )
    Input : \(\mathbf{G}_{p}^{d}\) is the full ghost layer 4.2).
    Result : if \(\mathrm{c} \in \mathcal{P}_{\mathfrak{p}}\) (or \(\overline{\mathcal{P}_{\mathfrak{p}}}\) ), leaf supp \({ }_{\mathrm{p}}\) (c) is computed and passed to callback.
    for all \(0 \leq t<K\) do
        \(\mathbf{G}^{t} \leftarrow \mathbf{G}_{\mathrm{p}}^{d} \cap \mathcal{O}^{t} \quad\) [subset of the ghost layer for tree \(t\left(O\left(\log \left|\mathbf{G}_{\mathrm{p}}^{d}\right|\right)\right)\) ]
        \(\mathbf{S}^{t} \leftarrow \mathbf{O}_{\mathrm{p}}^{t} \cup \mathbf{G}^{t} \quad\) [ \(\mathbf{O}_{\mathrm{p}}^{t}\) and \(\mathbf{G}^{t}\) are already ordered: no sorting is necessary]
    for all \(\mathrm{c} \in \bigcup_{0 \leq t<K} \mathrm{Clos}(\operatorname{root}(t))\) do
        for all \(0 \leq i<|\operatorname{supp}(\mathrm{c})|\) do
        \(t \leftarrow \operatorname{supp}(\mathrm{c})[i] \cdot t \quad\) [Each supp(c)[i] is the root of an octree]
            \(\mathrm{U}[i] \leftarrow \mathbf{S}^{t}\)
        Iterate_interior (c, U, callback)
```

Interate_interior operating on $c$ will only call a recursive copy for $e \in \operatorname{part}(c)$ if $\operatorname{dim}(\mathrm{e}) \geq d-1$.
6. A use case for the iterator: higher-order nodal basis construction. Thus far, we have developed algorithms for distributed forests with no special regard for numerical applications. In this section, we use our framework to perform a classic but complex task necessary for finite element computations, namely the globally unique numbering of degrees of freedom for a continuous finite element space over hanging-node meshes. We call it Lnodes in reference to (Gauß-)Lobatto nodes, which means that some nodes are located on element boundaries and are thus shared between multiple elements and/or processes, which presents some interesting challenges.

Hanging-node data structures have been discussed as early as 1980 22 and adapted effectively for higher-order spectral element computations [11, 24]. Specialpurpose data structures and interface routines have been defined for many discretization types built on top of octrees, including piecewise linear tensor-product elements [1,5] and discontinuous spectral elements [28]. The deal.II finite element software $\overline{3}$ uses yet another mesh interface [2]. In our presentation of Lnodes, we hope to show that the Iterate approach is sufficiently generic that it could be used to efficiently construct any of these data structures.
6.1. Concepts. In a hexahedral $n$-order nodal finite element, the Lagrangian basis functions and the degrees of freedom are associated with $(n+1)^{3} Q^{n}$-nodes located on a tensor grid of locations in the element. There is one node at each corner, $(n-1)$ nodes in the interior of each edge, $(n-1)^{2}$ nodes in the interior of each face, and $(n-1)^{3}$ nodes in the interior of the element, as in Figure 6.1 (left). If we endow each leaf in a forest of octrees with $Q^{n}$-nodes, we get $N \times(n+1)^{3}$ element nodes. $Q^{n}$-nodes are numbered lexicographically, and element-local nodes are then numbered to match the order of their associated leaves. The basis functions associated with the element nodes span a discontinuous approximation space.

We want to create a nodal basis for a $C^{0}$-conformal approximation space on $\Omega$ such that the restriction of the space to any leaf is spanned by the $Q^{n}$-nodes' basis functions. The nodes for the continuous basis functions are called global nodes. Each of the element nodes in the interior of a leaf can be associated with a unique global node, but on the boundary of a leaf, element nodes from multiple leaves may occupy the same location: in this case, the two element nodes are associated with the same global node, as in Figure 6.1 (middle). For non-conformal interfaces, the element nodes of the smaller leaves are not at the same locations as those of the larger leaf,


Fig. 6.1. (left) $Q^{n}$-nodes for $n=3$, with one node at each corner, $n-1$ nodes on (the interior of) each edge, $(n-1)^{2}$ nodes on each face, and $(n-1)^{3}$ nodes in each element. (middle, right) For both conformal and non-conformal interfaces, each element node corresponds to exactly one global node. (right) At non-conformal interfaces, an element may reference a global node remotely, as the small element references the top node.
but they cannot introduce new degrees of freedom, because every function in the space, when restricted to the non-conformal interface, must be representable using the larger leaf's basis functions. Conceptually, we can place the global nodes at the locations of the larger leaf's nodes and associate each element node from the smaller leaves with a single global node, as shown in Figure 6.1 (right). In reality, the value of a function at an element node on a non-conformal interface must be interpolated from the values at multiple global nodes, but the conceptual one-to-one association between a leaf's element nodes and global nodes is sufficient, in that it identifies all global nodes whose basis functions are supported on that leaf.

It is important to note that an element node of a leaf o may reference a global node that is contained in the domain of a point c that is outside the closure of $\operatorname{Dom}(\mathrm{o})$, and that $o$ is therefore not in the set of adjacent leaves leaf $\operatorname{supp}(c)$ defined in (5.4). In this situation, we say that o remotely references the global nodes in c. Formally, a leaf o remotely references a point c in the global partition set $\mathcal{P}_{\Omega}$ if

$$
\begin{equation*}
c \notin \operatorname{leaf} \operatorname{supp}(c) \text { and } \exists e \text { such that } o \in \operatorname{leaf} \operatorname{supp}(e) \text { and } c \in \operatorname{bound}(e) \tag{6.1}
\end{equation*}
$$

This relationship is also shown in Figure 6.1 (right). From this definition, we can see that the global nodes referenced by leaves in $\mathcal{O}_{\mathfrak{p}}$ will be contained in the closed locally relevant partition $\overline{\mathcal{P}_{\mathfrak{p}}} 5.3$. We note that a point c can be remotely referenced only if $\operatorname{dim}(\mathrm{c})<d-1$.
6.2. Data structures. On process $\mathfrak{p}$, we can represent the global nodes using an array $\mathbf{N}_{\mathfrak{p}}$ and the element nodes using a double array $\mathbf{E}_{\mathfrak{p}}$, where $\mathbf{E}_{\mathfrak{p}}[j][k]$ maps the $k$ th element node of $\mathcal{O}_{\mathfrak{p}}[j]$ to its global node in $\mathbf{N}_{\mathfrak{p}} . \mathbf{N}_{\mathfrak{p}}$ and $\mathbf{E}_{\mathfrak{p}}$ only reference locally relevant global nodes and thus implement fully distributed parallelism.

In presenting the Lnodes algorithm, we consider a global node $g$ to have the following data fields:

- g. $i$ : the globally unique index of this node,
- g.p: the process that owns g for the purposes of scatter/gather communication of node values,
- g. $\mathcal{S}_{\text {share }}$ : the sharer set of all processes that reference this node.

We include the sharer set $\mathcal{S}_{\text {share }}$ so that, in addition to the scatter/gather communication paradigm, the global nodes can also be used in the share-all paradigm, wherein any process that shares a node can send information about that node to all other processes that share that node $5^{5}$ If each process generates new information about

[^5]a node, the former paradigm requires two rounds of communication for information to disseminate, one gather and one scatter, while the latter requires one round, but with an increased number of messages. Each paradigm can be faster than the other, depending on communication latency, bandwidth, and other factors. We tend to place the highest weight on latency, hence our added support for share-all.

Remark 6.1. Most applications do not require higher-order finite element nodes, but the Lnodes data structure can be used in much more general settings. In particular, the Lnodes data structures for $n=2$ assign one unique global index to every point in $\overline{\mathcal{R}_{p}}$, and a map from each leaf to the points in its closure. If one symmetrizes these mappings, i.e., if one saves the leaf support sets leaf $\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})$ for $\mathrm{c} \in \overline{\mathcal{R}_{p}}$ generated by Iterate, then one has essentially converted the forest-of-octrees data structures into a graph-based unstructured mesh format with $O(1)$ local topology traversal. This format is typical of generic finite element libraries. Lnodes can therefore serve as the initial step in converting a forest of octrees into the format of an external library, with the remaining steps requiring little or no communication between processes.
6.3. Assigning global nodes. We want global nodes to be numbered independently of the number of processes $P$ and the partition of the leaves. For this reason, it is useful for each global node to be owned by one leaf, because a partition-independent order is then induced by combining lexicographic ordering of element nodes with the total octant order (see Algorithm 2.1). This computation is shown in Algorithm 6.1, which assumes that we have already determined which leaf $\mathcal{O}_{\mathfrak{p}}[j]$ owns each global node g , and temporarily stored that leaf's index in the global node's g. $i$ field.

```
Algorithm 6.1: Global_numbering (global node array \(\mathbf{N}_{\mathfrak{p}}\), double array \(\mathbf{E}_{\mathfrak{p}}\) )
    Input \(\quad: \forall \mathrm{g} \in \mathbf{N}_{\mathfrak{p}}, \underline{g} \cdot \mathfrak{p}=\mathfrak{q}\) such that the leaf that owns \(g\) is in \(\mathcal{O}_{\mathfrak{q}} ;\) if \(g \cdot p=\mathfrak{p}\), then \(g . \mathcal{S}_{\text {share }}\)
                is correct and g. \(i=j\) is (temporarily abused as) the index of the leaf that owns \(g\),
                \(\mathcal{O}_{\mathfrak{p}}[j]\).
    Result : correct global node data for all \(g \in \mathbf{N}_{p}\).
    integer array \(\mathbf{M}\left[\left|\mathbf{N}_{\mathfrak{p}}\right|\right] \quad\) [temporarily stores the local indices of global nodes]
    \(m \leftarrow 0 \quad\) [the number of global nodes owned by \(p\) ]
    for \(j=0\) to \(\left|\mathcal{O}_{\mathfrak{p}}\right|-1\) do
        for \(l=0\) to \((n+1)^{d}-1\) do
            \(k \leftarrow \mathbf{E}_{\mathfrak{p}}[j][l] \quad\) [index in \(\mathbf{N}_{\mathfrak{p}}\) of the global node referenced by this element node]
            if \(\mathbf{N}_{\mathfrak{p}}[k] \cdot \mathfrak{p}=\mathfrak{p}\) and \(\mathbf{N}_{\mathfrak{p}}[k] \cdot i=j\) then
                \(\mathbf{M}[k] \leftarrow m \quad\) [g's index among the locally owned nodes]
                \(m \leftarrow m+1\)
    \(\mathbf{t} \leftarrow\) Prefix_sums \((\operatorname{Allgather}(m)) \quad[\mathbf{t}[\mathfrak{q}]\) is the offset to the first node owned by \(\mathfrak{q}]\)
    for all \(0 \leq k<\left|\mathbf{N}_{\mathfrak{p}}\right|\) do
        \(\mathrm{g} \leftarrow \mathbf{N}_{\mathfrak{p}}[k]\)
        if \(\mathfrak{g} \cdot \mathfrak{p}=\mathfrak{p}\) then
            \(\mathrm{g} . i \leftarrow \mathbf{M}[k]+\mathbf{t}[\mathfrak{p}] \quad\) [all fields of g are now complete]
            send \(\mathfrak{g}\) to each \(\mathfrak{q} \in \mathrm{g} . \mathcal{S}_{\text {share }} \quad\) [in practice, grouped into one message per process]
        else
            receive updated \(g\) from \(g . p\)
```

We need a policy for assigning the ownership of nodes to leaves. We assign point $c \in \mathcal{P}_{\Omega}$ and its nodes to the first leaf o in leaf $\operatorname{supp}(c)$,

$$
\begin{equation*}
\text { owner (c) }:=\min \text { leaf } \operatorname{supp}(c) \tag{6.2}
\end{equation*}
$$

In the next subsection, we will show how this assignment policy allows for the global
nodes to be constructed without any more communication between processes than the communication in Global_numbering.
6.4. The Lnodes algorithm. The Lnodes algorithm (Algorithm 6.2) creates the global nodes and element nodes by iterating a callback Lnodes_callback over all points in the closed locally relevant partition $\overline{\mathcal{P}_{\mathfrak{p}}}$, which sets up a call to Global_numbering (Algorithm 6.1).

```
Algorithm 6.2: Lnodes (integer \(n\), ghost layer \(\mathbf{G}_{\mathfrak{p}}^{d}\) )
    Input : \(n>0\), the order of the nodes; the full ghost layer \(\mathbf{G}_{\mathrm{p}}^{d} 4.2\).
    Result : a double array \(\mathbf{E}_{\mathfrak{p}}\) of \(N_{\mathfrak{p}} \times(n+1)^{d}\) indices that maps element nodes to global
        nodes; an array \(\mathbf{N}_{\mathfrak{p}}\) of global nodes.
    global node array \(\mathbf{N}_{\mathfrak{p}} \leftarrow \emptyset\)
    integer double array \(\mathbf{E}_{\mathfrak{p}}\left[\left|\mathcal{O}_{\mathfrak{p}}\right|\right]\left[(n+1)^{d}\right]\)
    Iterate (Lnodes_callback, \(\mathbf{G}_{\mathfrak{p}}^{d}\) ) [initialize \(\mathbf{N}_{\mathfrak{p}}, \mathbf{E}_{\mathfrak{p}}\) ]
    Global_numbering ( \(\mathbf{N}_{\mathfrak{p}}, \mathbf{E}_{\mathfrak{p}}\) )
    [finalize \(\mathbf{N}_{p}\) ]
```

Given the assumptions of Global_numbering, Lnodes_callback has to accomplish the following for each global node g located at a point $\mathrm{c} \in \overline{\mathcal{P}_{\mathfrak{p}}}$ visited by Iterate:

1. determine the process g.p;
2. if $\mathfrak{g} \cdot \mathfrak{p}=\mathfrak{p}$, determine the index $j$ of owner(c) in $\mathcal{O}_{\mathfrak{p}}$;
3. if $\mathfrak{g} \cdot \mathfrak{p}=\mathfrak{p}$, determine the set $\mathfrak{g} \cdot \mathcal{S}_{\text {share }}$ of processes that share $\mathfrak{g}$;
4. complete the entries in $\mathbf{E}_{\mathfrak{p}}$ that refer to g .

We will not give pseudocode for Lnodes_callback here; we only hope to convince the reader that the information that Iterate supplies to the Lnodes_callback-i.e., the local leaf support set leaf $\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})$ for each point c in the closed locally relevant partition $\overline{\mathcal{P}_{\mathfrak{p}}}$-is sufficient to accomplish the above listed tasks.

1. Determine g.p. The policy that defines owner(c) 6.2 guarantees that c and its nodes will be owned by a leaf in the partition of the first process $\mathfrak{q}$ such that $\overline{\Omega_{\mathfrak{q}}}$ intersects $\operatorname{Dom}(c)$. This means that each process $\mathfrak{p}$ that references can determine the processes that own all of the nodes it references, even if leaf $\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})$ is incomplete (see Algorithm 6.3).
```
Algorithm 6.3: Determine_owner_process (point c)
    Result : if \(\mathrm{c} \in \mathcal{P}_{\Omega}\), the process that own's c and its global nodes.
    for \(\mathrm{e} \in\{\mathrm{c}\} \cup \operatorname{part}(\mathrm{c})\) such that \(\operatorname{dim}(\mathrm{e})=0\) do \(\quad\) [this set always contains one 0-point]
        return \(\min \{\) locate(a) \(: \mathrm{a} \in\) atom \(\operatorname{supp}(\mathrm{e}))\}\)
```

2. If $\mathrm{g} \cdot \mathfrak{p}=\mathfrak{p}$, determine owner(c). Suppose owner $(\mathrm{c})=\mathrm{o} \in \mathcal{O}_{\mathfrak{p}}$ : $\circ$ and c intersect, $\overline{\operatorname{Dom}(o)} \cap \operatorname{Dom}(\mathrm{c}) \neq \emptyset$. By definition (5.4), c must be in $\mathcal{P}_{\mathfrak{p}}$, so by proposition 5.2, leaf $\operatorname{supp}_{\mathfrak{p}}(c)=$ leaf $\operatorname{supp}(c)$. Therefore, owner(c) will be in leaf $\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})$, and its index in $\mathcal{O}_{\mathfrak{p}}$ was calculated by Iterate, so we can set g. $i$ equal to that index for each global node $g$ located at c .
3. If $\mathfrak{g} \cdot \mathfrak{p}=\mathfrak{p}$, determine $\mathrm{g} \cdot \mathcal{S}_{\text {share }}$. We use both Proposition 5.2 and the $2: 1$ balance condition to design an algorithm called Reconstruct_remote to reconstruct the octant data for all leaves that remotely reference 6.1 the point c and its nodes (Algorithm 6.4. Figure 6.2. By locating the processes that overlap the octants in leaf $\operatorname{supp}_{\mathfrak{p}}(c)$ and the octants returned by Reconstruct_remote(c, leaf $\operatorname{supp}_{\mathfrak{p}}(c)$ ), process $\mathfrak{p}$ can determine all processes that reference c's nodes.


Fig. 6.2. Because of the 2:1 condition, Reconstruct_remote (Algorithm 6.4) can use the octants in leaf $\operatorname{supp}(c)$ (left) to reconstruct remotely referencing leaves (right).

```
Algorithm 6.4: Reconstruct_remote (point c, octant set leaf supp(c))
    Input : the leaf support set leaf supp(c) 5.4.
    Result : the set \(\mathcal{R}\) of all octants that are leaves that remotely reference c .
    \(\mathcal{R} \leftarrow \emptyset\)
    for all \(\mathrm{e} \in \bigcup\) bound(leaf supp(c)) do [for every leaf's boundary point \(\ldots\) ]
        if \(\mathrm{c} \in\) bound(e) then [...that is adjacent to c ]
                for all \(\mathbf{s} \in \bigcup \operatorname{supp}(\operatorname{child}(\mathbf{e}))\) do \(\quad\) [for every octant \(\mathbf{s}\) adjacent to a child of e]
                    if not \(\mathrm{s} \in \bigcup\) desc(leaf supp(c)) then [if \(\mathbf{s}\) is not a descendant of a leaf ]
                    \(\mathcal{R} \leftarrow \mathcal{R} \cup\{\mathrm{s}\} \quad\) [s remotely references c 6.1]]
    return \(\mathcal{R}\)
```

4. Complete the references to g in $\mathbf{E}_{\mathfrak{p}}$. For each leaf $\circ \in \operatorname{leaf} \operatorname{supp}_{\mathfrak{p}}(\mathrm{c}) \cap \mathcal{O}_{\mathfrak{p}}$ Iterate provides the index $j$ in $\mathcal{O}_{\mathfrak{p}}$ such that $\mathcal{O}_{\mathfrak{p}}[j]=0$, so determining the values of $k$ such that $\mathbf{E}_{\mathfrak{p}}[j][k]$ refer to the global nodes at point c is a matter of comparing the orientation of $o$ and $c$ relative to each other. For each leaf $r \in \mathcal{O}_{\mathfrak{p}}$ that remotely references c and its nodes, by definition (6.1) there must be a point e such that $\mathrm{c} \in \operatorname{bound}(\mathrm{e})$ and $\mathrm{r} \in$ leaf $\operatorname{supp}(\mathrm{e})$. The instance of Lnodes_callback called for the point e has the index $j$ such that $\mathcal{O}_{\mathrm{p}}[j]=\mathrm{r}$ and can "link" it to c , so that the correct $\mathbf{E}_{\boldsymbol{p}}[j][\star]$ entries for c's nodes can be completed (see Remark 5.3 ).

The previously presented algorithm Nodes [6, Algorithm 21] produces data structures equivalent to those produced by Lnodes for $n=1$. The ownership rule in Nodes - associating each node with a unique level- $\left(l_{\max }+1\right)$ octant, and assigning ownership based on the process whose range contains that octant - is in principle the same as the ownership rule given in (6.2). Nodes does not have symmetric communication, however, because it does not construct the neighborhood leaf $\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})$ when it creates a node at c, and so it cannot perform a calculation like Reconstruct_remote. Because it does not deduce the presence of remotely-sharing processes, Nodes requires a handshaking step, where the communication pattern is determined.
7. Performance evaluation. In this section we evaluate the efficiency and scalability of the algorithms presented in this work as they have been implemented in p4est. The parallel scalability is assessed on the Blue Gene/Q supercomputer JUQUEEN, which is configured with 28,672 compute nodes, each with 16 GB of memory and 16 cores, for a total of 458,752 cores. Additional concurrency is available


Fig. 7.1. Example forests of octrees for nontrivial domain topologies. (left) A cutaway of a shell geometry, composed of 24 mapped octrees. (right) A collection of six rotated and mapped octrees connected in an irregular topology. Both show adhoc refinement patterns; a 2:1-balance condition has been enforced in the left hand plot, but not the right. Color indicates the partitions of different processes.
through simultaneous multithreading: each core has two instruction/integer pipelines, and can issues one instruction to each of these pipelines per cycle, as long as they come from different threads. Where appropriate, we will compare results for 16,32 , and 64 MPI processes per node (p4est uses only MPI for parallelism). We have compiled the p4est library and executables with IBM's XL C compiler in version 12.1.
7.1. Search. To test the performance of Search (Algorithm3.1), we consider the problem of identifying the leaves that contain a set of randomly generated points. We choose a spherical shell domain typical for simulations of earth's mantle convection, with inner radius $r=0.55$ and outer radius $R=1$, as illustrated in Figure 7.1 (left). For each test, we generate $M$ points at random, independently and uniformly distributed in the cube containing the shell, and use Search as implemented by the p4est function p8est_search to identify the leaves that contain them.

The mappings $\varphi^{t}$ for the octrees (2.13) are given analytically for this domain. In the callback that we provide to Search, we have two tests to determine whether the mapped domain $\operatorname{Dom}(\mathrm{o})$ of an octant o contains a point $x$, one fast and inaccurate in the sense of allowing false positive matches, the other slower but accurate. In the accurate test, the mapping $\varphi^{t}$ is inverted to get the preimage $\xi$ of $x$, and a bounding box calculation determines whether $\xi \in \operatorname{dom}(o)$. In the inaccurate test, the image $x_{\mathrm{o}}$ of the octant's center is computed, as well as an upper-bound $r_{\circ}$ on the radius of the bounding sphere of $\operatorname{Dom}(\mathrm{o})$, and we test whether $\left|x-x_{o}\right| \leq r_{o}$. In Search the accurate test is performed only when o is a leaf. We perform our tests on a series of forests with increasing numbers of leaves $N$, but with each forest refined so that the finest leaves are four levels more refined than the coarsest.

In Figure 7.2, we present the scaling results for our tests. Each of the $P$ processes must determine which of the $M$ points are in its partition. This means that each process must perform the inaccurate test at least $M$ times. This is why, for fixed values of $P$, we see a scaling with $O(M)$. Indeed, the fraction of points that fall in a given processes partition is on average $1 / P$, so for large values of $P$ the majority of the runtime is spent on points that are not in the partition. This is why, in Figure 7.2 , the number of leaves on a node $N / P$ has so little effect on the runtime. When we
$M$ searches for one point, runtime in seconds

One search for $M$ points, runtime in seconds



$$
\begin{array}{|ll|}
\hline+P_{0}, & D_{0} \\
\times P_{0}, & D_{1} \\
+P_{1}, & D_{0} \\
\times P_{1}, & D_{1} \\
\otimes P_{1}, & D_{2} \\
+P_{2}, & D_{0} \\
\times P_{2}, & D_{1} \\
\otimes P_{2}, & D_{2} \\
\hline
\end{array}
$$

Fig. 7.2. Scaling results for searching for $M$ points in a shell domain using Search as implemented by p8est_search. We examine various values of $M, P$, and $N$. Three different values of $P$ are used: $P_{0}=64, P_{1}=64^{2}$, and $P_{2}=64^{3}=262144$. Three different values of $N / P=D$ are used: $D_{0} \approx 15 \mathrm{k}, D_{1} \approx 122 \mathrm{k}$, and $D_{2} \approx 979 \mathrm{k}$. (left) $M$ separate calls of Search are used to find the $M$ points. (right) One call of Search is used to find all $M$ points. The dotted lines symbolize linear weak scaling; points on top of each other demonstrate the independence of the runtime from the local octant count $D_{i}$. The largest number of octants reached is $2.568 \times 10^{11}$.
take advantage of the algorithm's ability to search for multiple points simultaneously, however, the setup costs of the inaccurate test, such as computing the bounding radius $r_{o}$, can be amortized over multiple comparisons. Hence we see significant speedup when searching for multiple points simultaneously: in Figure 7.2 , we see that for large values of $P$ and for $M / P \geq 1$ the simultaneous search is roughly 64 times faster than searching for the same points individually.
7.2. Ghost. We test the performance of ghost layer construction as implemented by the p4est function p8est_ghost, on the irregular geometry shown in Figure 7.1 (right). We again create a series of meshes with increasing $N$ and four levels of difference between the coarsest and finest leaves. Our ghost layer construction uses Find_range_boundaries (Algorithm 4.2) to determine which processes' partitions border an octant o. When a partition $\Omega_{\mathfrak{p}}$ with $N_{\mathfrak{p}}$ leaves is well-shaped, we expect $O\left(N_{\mathfrak{p}}^{2 / 3}\right)$ of those leaves to be on the boundary of $\Omega_{\mathfrak{p}}$, so we consider $O\left((N / P)^{2 / 3}\right)$ to be ideal scaling. In the algorithm Add_ghost (Algorithm4.1), which is called for each boundary leaf, $\log (P)$ work is performed to determine a set of potentially neighboring processes; the remaining leaves in the interior of the domain are skipped without calling Add_ghost, and so they should contribute very little to the runtime of Ghost. We therefore expect our performance to be $O\left((N / P)^{2 / 3} \log P\right)$.

In Figure 7.3 , we give plots for assessing the strong- and weak-scalability of ghost layer construction, relative to the ideal $O\left((N / P)^{2 / 3}\right)$ scaling. (For almost all problems, assigning 64 processes per node was slower than 32 , so we omit this data from the figure.) We see good strong-scalability for $16 \leq P \leq 65 \mathrm{k}$ and $N / P \geq 1 \mathrm{k}$. For the full machine, when $P=459 \mathrm{k}$ and $P=918 \mathrm{k}$, the communication latency and the small amount of $O(P)$ workspace and work in the implementation (two scans of arrays of 32 -bit integers) limits the efficiency for $N / P \leq 10 \mathrm{k}$. In the weak-scaling plot, for the largest values of $N / P$, we see that the relative efficiency (the efficiency of $(8 N, 8 P)$

Ghost runtime in seconds


Ghost runtime in secs. $/(N / P)^{2 / 3}$


| $P$, 16-way |  | $\triangle 128$ | $\square 1024$ | $\diamond 8192$ | $\bigcirc 65536$ |  | 458752 |
| :---: | :---: | :---: | :---: | :---: | :---: | :---: | :---: |
| $P$, 32-way | - 32 | $\triangle 256$ | $\square 2048$ | -16384 | -131072 |  | 917504 |

Fig. 7.3. The scalability of ghost layer construction. The meshes used are described in the text: the largest number of leaf octants $N$ is $5.1 \times 10^{11}$. (right) Runtime as a function of $P$ : for 16-way process distribution, we compare strong scaling (solid lines) to ideal $O\left((N / P)^{2 / 3}\right)$ scaling (dotted). The total number of leaves $N$ in each mesh is indicated. (right) runtime scaled by $(N / P)^{2 / 3}$ as a function of $N / P$. Weak-scaling is assessed by comparing the vertical distance between points: each grid line represents a $25 \%$ loss of weak-scaling efficiency relative to the ideal $O\left((N / P)^{2 / 3}\right)$.
relative to $(N, P)$ ) improves slightly as $N$ and $P$ increase, which suggests that we are seeing $O\left((N / P)^{2 / 3} \log P\right)$ scaling asymptotically.
7.3. Serial comparison of Lnodes and Nodes. For polynomial degree $n=$ 1, the data structures constructed by Nodes [6, Algorithm 21] and Lnodes (Algorithm 6.2 are essentially equivalent. For a general forest of octrees on a single process, both have $\mathcal{O}(N \log N)$ runtimes. While Nodes uses repeated binary searches and hash table queries and insertions, Lnodes uses Iterate (Algorithm 5.3) to recursively split the forest and operates on subsets of leaves. This divide-and-conquer approach should make better use of a typical cache hierarchy. In this subsection, we present a small experiment that confirms this fact.

The experiment is conducted on a single octree using a single process. We again create a series of meshes with increasing $N$ and four levels of difference between the coarsest and finest leaves. For each forest in the series, we have three programs: one that calls Nodes, one that calls Lnodes, and one that calls neither. We use the Linux utility perf ${ }^{6}$ to estimate the number of instructions, cache misses, and branch prediction misses in each program, calling each program 30 times to compensate for the noise in perf's sampling. The averages of the events from the program calling neither routine is subtracted from the other two averages, giving an estimate of the events that can be attributed to the two routines.

The experiment is performed on a laptop with two Intel Ivy Bridge Core i7-3517U dual core processors. Each core has a 64 kB on-chip L1 cache, a 256 kB L2 cache, and each processor has a 4 MB L3 cache: perf counts L3 cache misses. The p4est library and the executable are compiled by gcc 4.6 .4 with -03 optimization.

[^6]Table 7.1
Serial performance comparison of Nodes (top) and Lnodes for $n=1$ (bottom), as implemented by the p4est functions p8est_nodes and p8est_lnodes, on a series of single-octree forests.

| $N$ | runtime $(\mathrm{ms})$ | instructions | branch misses | cache misses |
| :--- | :--- | :--- | :--- | :--- |
| $4.6 \times 10^{3}$ | $9.5 \times 10^{0}$ | $4.3 \times 10^{7}$ | $2.1 \times 10^{5}$ | $2.2 \times 10^{4}$ |
|  | $1.0 \times 10^{1}$ | $3.7 \times 10^{7}$ | $5.3 \times 10^{4}$ | $1.1 \times 10^{4}$ |
| $3.9 \times 10^{4}$ | $8.6 \times 10^{1}$ | $4.2 \times 10^{8}$ | $1.7 \times 10^{6}$ | $2.2 \times 10^{5}$ |
|  | $4.0 \times 10^{1}$ | $3.1 \times 10^{8}$ | $3.6 \times 10^{5}$ | $5.1 \times 10^{4}$ |
| $3.2 \times 10^{5}$ | $8.4 \times 10^{2}$ | $3.7 \times 10^{9}$ | $1.3 \times 10^{7}$ | $4.8 \times 10^{6}$ |
|  | $3.5 \times 10^{2}$ | $2.5 \times 10^{9}$ | $2.7 \times 10^{6}$ | $4.5 \times 10^{5}$ |
| $2.6 \times 10^{6}$ | $8.0 \times 10^{3}$ | $3.3 \times 10^{10}$ | $1.0 \times 10^{8}$ | $6.1 \times 10^{7}$ |
|  | $2.8 \times 10^{3}$ | $2.0 \times 10^{10}$ | $2.2 \times 10^{7}$ | $5.4 \times 10^{6}$ |

The results of the experiment are given in Table 7.1. The table shows that the advantages of Lnodes over Nodes in terms of the number of instructions and the number of branch misses do not grow much with $N$, but the advantage in terms of cache misses grows from a factor of 2 on the smallest problem size to a factor of 11 on the largest.
7.4. Parallel scalability of Lnodes. In the previous subsection we compared the per-process efficiency of Lnodes and Nodes. Here we compare their parallel scalability on the same series of test forests used to test Ghost above.

In Figure 7.4 (top), we show the runtimes of Lnodes for $n=1$. As discussed in Section 5.4, the implementation of Iterate has been optimized for large values of $N / P$ : this optimization requires $O\left(l_{\max }\right)$ workspace and setup time. The weak-scaling plot shows that the optimization is effective, in that for $N / P \geq 10 \mathrm{k}$ and $P \leq 262 \mathrm{k}$ the weak-scalability is nearly ideal, and that the absolute runtime is small ( $\sim 20$ seconds for $1 M$ leaves/process). The optimization requires redudant work, however, and this affects the efficiency for $N / P<1 \mathrm{k}$. The strong-scaling plot shows good scalability for $P \leq 262 \mathrm{k}$ and $(N / P)>1 \mathrm{k}$, and in this range the algorithm benefits from 32 and 64 processes per node as well. As in the scaling for ghost layer construction, the communication latency and the small amount of $O(P)$ work in the implementation finally limit the scalability for the smallest meshes that were timed on the full machine.

In the same figure (bottom left) we compare the runtimes for Lnodes for $n=1$ to the runtimes of Nodes [6, Algorithm 21]. For most tests, Lnodes is faster than Nodes: although the relative advantage is smaller on the Blue Gene/Q architecture of JUQUEEN than on the Ivy Bridge architecture used in the serial test, we still see the advantage increasing as $N / P$ increases, which is suggestive of better cache performance. The communication pattern of Lnodes, consisting of one allgather and one round of point-to-point communication, is more scalable than the communication pattern of Nodes, which includes a handshake component, hence the better performance of Lnodes for small values of $N / P$ and large values of $P$.

Finally, Figure 7.4 also compares the scalability of Lnodes for higher polynomial orders to the scalability for $n=1$ (bottom right). We see that the runtime to construct 7 th-order nodes is never more than six times the runtime to construct 1 st-order nodes, even though there are 64 times as many element nodes and roughly 500 times as many

Lnodes ( $n=1$ ) runtime in seconds


Lnodes ( $n=1$ ) runtime in secs. $/(N / P)$



Nodes runtime / Lnodes ( $n=1$ ) runtime

Lnodes ( $n=7$ ) runtime /
Lnodes ( $n=1$ ) runtime



FIG. 7.4. The parallel scalability of the Lnodes algorithm, as implemented by the p4est function p8est_lnodes. (top) Runtimes for $n=1$. (top left) Runtime as a function of $P$, comparing strong scaling (solid lines) to ideal $O(N / P)$ scaling (dotted). The total number of leaves $N$ in each mesh is indicated. (top right) Runtime scaled by $N / P$ as a function of $N / P$. Weak-scaling is assessed by comparing the vertical distance between points: each grid line represents a $25 \%$ loss of weak-scaling efficiency. (bottom left) The speedup of Lnodes versus Nodes as implemented by p8est_nodes is shown for the same meshes as above. (bottom right) The runtime for $n=7$, scaled by the runtime for $n=1$.
global nodes $\sqrt[7]{7}$ For large values of $P$ the communication costs, which do not increase significantly with $n$, dominate the runtime, so that the cost of constructing high-order

[^7]nodes is essentially the same as 1st-order nodes.
8. Conclusion. In this work, we introduce new recursive algorithms that operate on the distributed forest-of-octrees data structures that the p4est software defines and uses to support scalable parallel AMR. The algorithms developed here exploit a recursive space partition from a topological point of view. They constitute p4est's high-level reference interface, which is designed to be used directly from third-party numerical applications.

With the Search algorithm, we demonstrate how to efficiently traverse a linear octree downward from the root, even though the flat storage of leaves has no explicit tree structure. This search operation is in some sense purely hierarchical: a similar search could be performed even if the nodes and leaves of the tree were not interpreted as a space partition in $\mathbb{R}^{d}$.

As a component of the Ghost algorithm, we propose a recursive algorithm for determining the intersections between lower-dimensional boundary cubes and ranges of leaves that are specified only by the first and last leaves in the range. This algorithm is notable in that, while the procedure is recursive on the implicit octree structure, the result that it computes - a set of intersections-is purely topological in nature.

In the Iterate algorithm, we present a method of performing callback-based iteration over leaves and leaf boundaries that construct local topological information for the callback on the fly. This procedure combines aspects of the two previous algorithms: it involves recursion over the octree hierarchy and recursion over topological dimension. The divide-and-conquer nature of the algorithm makes better use of the cache hierarchy than approaches to iteration that rely on repeated searches through the array of leaves, as we demonstrate in practice.

We use Iterate in the construction of fully-distributed higher-order $C^{0}$ finite element nodes in the algorithm Lnodes. The topological information provided by Iterate simplifies the handling of non-conformal interfaces, and provides sufficient information to allow for node assignments to be made without communication, and for the communication pattern between referencing processes to be determined without handshaking. In practice, this gives us good scalability, which we have demonstrated to nearly a half million cores on the JUQUEEN supercomputer. The implementation has been tuned for granularities of a thousand leaves per MPI process and above, and in this range we good scalability, although room for improvement remains for smaller granularities.

The scalability of Lnodes that we have demonstrated is important for more applications than just higher-order finite element nodes, because the data structures returned by the Lnodes algorithm can also serve as the basis for converting a linear forest of octrees into an unstructured mesh adjacency graph. Lnodes includes all of the communication necessary for this conversion, so the same scalability should be achievable by third-party numerical codes that use Lnodes (or a similar approach based on Iterate) to interface p4est with their own mesh formats.

Reproducibility. The algorithms presented in this article are implemented in the p4est reference software [4]. p4est, including the programs used in the performance analysis presented above, is free and freely downloadable software published under the GNU General Public License version 2, or (at your option) any later version.

Acknowledgments. The first author thanks the U.S. Department of Energy for support by the Computational Science Graduate Fellowship (DOE CSGF) and by the Office of Science (DOE SC), Advanced Scientific Computing Research (ASCR),

Scientific Discovery through Advanced Computing (SciDAC) program, under award number DE-FG02-09ER25914. The second author is supported by the German Research Foundation (DFG)'s Transregio 32 initiative.

The authors gratefully acknowledge the Gauß Centre for Supercomputing (GCS) for providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS share of the supercomputer JUQUEEN at Jülich Supercomputing Centre (JSC). GCS is the alliance of the three national supercomputing centres HLRS (Universität Stuttgart), JSC (Forschungszentrum Jülich), and LRZ (Bayerische Akademie der Wissenschaften), funded by the German Federal Ministry of Education and Research (BMBF) and the German State Ministries for Research of Baden-Württemberg (MWK), Bayern (StMWFK) and Nordrhein-Westfalen (MIWF).

The authors are indebted to three anonymous reviewers, whose remarks led to significant improvements in the final form of this paper, and to Jose A. Fonseca and Johannes Holke for their editorial help.

## REFERENCES

[1] V. Akçelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. R. O'Hallaron, T. Tu, and J. Urbanic, High resolution forward and inverse earthquake modeling on terascale computers, in SC03: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM/IEEE, 2003. Gordon Bell Prize for Special Achievement.
[2] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, Algorithms and data structures for massively parallel generic adaptive finite element codes, ACM Transactions on Mathematical Software, 38 (2011), pp. 14:1-14:28.
[3] W. Bangerth, R. Hartmann, and G. Kanschat, deal.II - a general-purpose object-oriented finite element library, ACM Transactions on Mathematical Software, 33 (2007), p. 24.
[4] C. Burstedde, p4est: Parallel AMR on forests of octrees, 2010. http://www.p4est.org/.
[5] C. Burstedde, G. Stadler, L. Alisic, L. C. Wilcox, E. Tan, M. Gurnis, and O. Ghattas, Large-scale adaptive mantle convection simulation, Geophysical Journal International, 192 (2013), pp. 889-906.
[6] C. Burstedde, L. C. Wilcox, and O. Ghattas, p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees, SIAM Journal on Scientific Computing, 33 (2011), pp. 1103-1133.
[7] P. Colella, J. Bell, N. Keen, T. J. Ligocki, M. Lijewski, and B. V. Straalen, Performance and scaling of locally-structured grid methods for partial differential equations, Journal of Physics: Conference Series, 78 (2007), pp. 1-13.
[8] P. Colella, D. T. Graves, N. Keen, T. J. Ligocki, D. F. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen, Chombo Software Package for AMR Applications. Design Document., Applied Numerical Algoirthms Group, NERSC Division, Lawrence Berkeley National Laboratory, Berkeley, CA, May 2007.
[9] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, Zoltan data management services for parallel dynamic applications, Computing in Science and Engineering, 4 (2002), pp. 90-97.
[10] J. Dreher and R. Grauer, Racoon: A parallel mesh-adaptive framework for hyperbolic conservation laws, Parallel Computing, 31 (2005), pp. 913-932.
[11] P. F. Fischer, G. W. Kruse, and F. Loth, Spectral element methods for transitional flows in complex geometries, Journal of Scientific Computing, 17 (2002), pp. 81-98.
[12] T. Goodale, G. Allen, G. Lanfermann, J. Masso, T. Radke, E. Seidel, and J. Shalf, The Cactus framework and toolkit: Design and applications, in Vector and Parallel Processing - VECPAR '2002, 5th International Conference, Springer, 2003.
[13] R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, et al., The IBM Blue Gene/Q compute chip, Micro, IEEE, 32 (2012), pp. 48-60.
[14] T. Isaac, C. Burstedde, and O. Ghattas, Low-cost parallel algorithms for 2:1 octree balance, in Proceedings of the 26th IEEE International Parallel \& Distributed Processing Symposium, IEEE, 2012.
[15] Forschungszentrum Jülich - JUQUEEN. http://www.fz-juelich.de/ias/jsc/EN/Expertise/ Supercomputers/JUQUEEN/JUQUEEN_node.html Last accessed May 2, 2014.
[16] O. S. Lawlor, S. Chakravorty, T. L. Wilmarth, N. Choudhury, I. Dooley, G. Zheng, and L. V. Kalé, ParFUM: a parallel framework for unstructured meshes for scalable dynamic physics applications, Engineering with Computers, 22 (2006), pp. 215-235.
[17] J. Luitjens, M. Berzins, and T. C. Henderson, Scalable parallel AMR for the Uintah multiphysics code, in Petascale Computing Algorithms and Applications, D. A. Bader, ed., Chapman \& Hall, 2008, pp. 67-81.
[18] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer, Paramesh: A parallel adaptive mesh refinement community toolkit, Computer physics communications, 126 (2000), pp. 330-354.
[19] J. P. May, A concise course in algebraic topology, University of Chicago Press, 1999.
[20] G. M. Morton, A computer oriented geodetic data base; and a new technique in file sequencing, tech. rep., IBM Ltd., 1966.
[21] C. D. Norton, J. Z. Lou, and T. A. Cwik, Status and directions for the PYRAMID parallel unstructured $A M R$ library, in Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2001, p. 120.
[22] W. C. Rheinboldt and C. K. Mesztenyi, On a data structure for adaptive finite element mesh refinements, ACM Transactions on Mathematical Software, 6 (1980), pp. 166-187.
[23] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros, Dendro: Parallel algorithms for multigrid and $A M R$ methods on 2:1 balanced octrees, in SC'08: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM/IEEE, 2008.
[24] C. Sert and A. Beskok, Spectral element formulations on non-conforming grids: A comparative study of pointwise matching and integral projection methods, Journal of Computational Physics, 211 (2006), pp. 300-325.
[25] J. R. Stewart and H. C. Edwards, A framework approach for developing parallel adaptive multiphysics applications, Finite Elements in Analysis and Design, 40 (2004), pp. 15991617.
[26] H. Sundar, R. Sampath, and G. Biros, Bottom-up construction and 2:1 balance refinement of linear octrees in parallel, SIAM Journal on Scientific Computing, 30 (2008), pp. 2675-2708.
[27] T. Tu, D. R. O'Hallaron, and O. Ghattas, Scalable parallel octree meshing for terascale applications, in SC '05: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, ACM/IEEE, 2005.
[28] L. C. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas, A high-order discontinuous Galerkin method for wave propagation through coupled elastic-acoustic media, Journal of Computational Physics, 229 (2010), pp. 9373-9396.

Appendix A. Proof of the correctness of Find_range_boundaries (Algorithm (4.2).

THEOREM A.1. Given a range $[\mathrm{f}, \mathrm{l}]$, where f and l are atoms with a common ancestor $\mathbf{s}$, and given a set of boundary indices $\mathcal{B}_{\text {query }} \subseteq \mathcal{B}$, Algorithm 4.2 returns the set $\mathcal{B}_{\cap}(\mathrm{f}, 1, \mathrm{~s}) \cap \mathcal{B}_{\text {query }}$ (4.4).

Proof. The proof is inductive on the refinement level, s.l.
If $\mathrm{s} . l=l_{\text {max }}$, then the only descendant of s is itself, so $\mathrm{s}=\mathrm{f}=1$. Therefore $\overline{\bigcup \operatorname{Dom}([f, l])}=\overline{\operatorname{Dom}(s)}$ and $\mathcal{B}_{\cap}(f, l, s)=\mathcal{B}$, so $\mathcal{B}_{\cap}(f, l, s) \cap \mathcal{B}_{\text {query }}=B_{\text {query }}$. This is correctly returned on line 1

Now suppose that Find_range_boundaries returns correctly if $l_{\max } \geq \mathrm{s} . l \geq m$, and suppose $\mathbf{s} . l=m-1$. Let $\mathbf{f} \in \operatorname{desc}(\operatorname{child}(\mathbf{s})[j])$ and $l \in \operatorname{desc}(\operatorname{child}(\mathbf{s})[k])$ : the set $\mathcal{I}$ defined in 4.7) is equal to $\{j, \ldots, k\}$.

If $j=k$, then by proposition 4.1, $\mathcal{B}_{\cap}(\mathbf{f}, 1, \mathbf{s})=\mathcal{B}_{\cap}(\mathbf{f}, 1, \operatorname{child}(\mathbf{s})[j]) \cap \mathcal{B}_{\cap}^{j}$. By the inductive assumption, line 4 returns

$$
\begin{equation*}
\mathcal{B}_{\cap}(\mathbf{f}, 1, \operatorname{child}(\mathbf{s})[j]) \cap\left(\mathcal{B}_{\cap}^{j} \cap \mathcal{B}_{\text {query }}\right)=\mathcal{B}_{\cap}(\mathbf{f}, 1, \mathbf{s}) \cap \mathcal{B}_{\text {query }} . \tag{A.1}
\end{equation*}
$$

Now suppose $j<k$. If the range $[\mathrm{f}, \mathrm{l}]$ overlaps all of child $(\mathbf{s})[i]$, then the set $\mathcal{B}_{\cap}\left(\mathrm{f}_{i}, 1_{i}, \operatorname{child}(\mathbf{s})[i]\right)$ is equal to $\mathcal{B}$. This is the case if $j<i<k$, so the set $\mathcal{B}_{\text {match }}^{1}$
computed on line 5 is

$$
\begin{equation*}
\mathcal{B}_{\text {match }}=\bigcup_{j<i<k} \mathcal{B}_{\text {query }} \cap \mathcal{B}_{\cap}^{i}=\bigcup_{j<i<k} \mathcal{B}_{\text {query }} \cap\left(\mathcal{B}_{\cap}\left(\mathbf{f}_{i}, 1_{i}, \operatorname{child}(\mathbf{s})[i]\right) \cap B_{\cap}^{i}\right) \tag{A.2}
\end{equation*}
$$

This is also the case for $i=j$ if $\mathrm{f}=\mathrm{f}_{j}$, so on each branch of the condition on line 8 the set $\mathcal{B}_{\text {match }}^{j}$ is computed as

$$
\begin{align*}
\mathcal{B}_{\text {match }}^{j} & =\mathcal{B}_{\cap}\left(\mathbf{f}_{j}, 1_{j}, \operatorname{child}(\mathbf{s})[j]\right) \cap\left(\left(\mathcal{B}_{\text {query }} \cap \mathcal{B}_{\cap}^{j}\right) \backslash \mathcal{B}_{\text {match }}\right)  \tag{A.3}\\
& =\left(\mathcal{B}_{\text {query }} \cap\left(\mathcal{B}_{\cap}\left(\mathbf{f}_{j}, 1_{j}, \operatorname{child}(\mathbf{s})[j]\right) \cap \mathcal{B}_{\cap}^{j}\right)\right) \backslash \mathcal{B}_{\text {match }} .
\end{align*}
$$

By the same reasoning, on each branch of the conditional on line 11 , the set $\mathcal{B}_{\text {match }}^{k}$ is computed as

$$
\begin{equation*}
\mathcal{B}_{\text {match }}^{k}=\left(\mathcal{B}_{\text {query }} \cap\left(\mathcal{B}_{\cap}\left(\mathrm{f}_{k}, 1_{k}, \operatorname{child}(\mathbf{s})[k]\right) \cap \mathcal{B}_{\cap}^{k}\right)\right) \backslash \mathcal{B}_{\text {match }} \backslash \mathcal{B}_{\text {match }}^{j} \tag{A.4}
\end{equation*}
$$

The union $\mathcal{B}_{\text {match }} \cup \mathcal{B}_{\text {match }}^{j} \cup \mathcal{B}_{\text {match }}^{k}$ is therefore equal to

$$
\begin{equation*}
\mathcal{B}_{\text {query }} \cap \bigcup_{j \leq i \leq k} \mathcal{B}_{\cap}\left(\mathbf{f}_{i}, 1_{i}, \operatorname{child}(\mathbf{s})[i]\right) \cap \mathcal{B}_{\cap}^{i}=\mathcal{B}_{\text {query }} \cap \mathcal{B}_{\cap}(\mathbf{f}, 1, \mathbf{s}) . \tag{A.5}
\end{equation*}
$$

By induction, the proof is complete.
Appendix B. Proof of the correctness of Iterate_interior (Algorithm 5.2).

Let the definitions in Section 5 be given. We prove the correctness of Iterate_interior (Algorithm 5.2 when the relevant set is $\overline{\mathcal{P}_{\mathfrak{p}}}$. The proof for the case when $\mathcal{P}_{\mathfrak{p}}$ is the relevant set is very similar.

Theorem B.1. Assume that the requirements for the arguments of Algorithm 5.2 are met. If $\mathrm{c} \in \overline{\mathcal{P}_{\mathfrak{p}}}$, then leaf $\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})$ is correctly computed. If there is a subset of $\overline{\mathcal{P}_{\mathfrak{p}}}$ whose domain is contained in $\operatorname{Dom}(\mathrm{c})$, then the callback function is executed for all points in that subset.

Proof. We first assert that if $c \in \overline{\mathcal{P}_{\mathfrak{p}}}$ then its local leaf support leaf $\operatorname{supp}_{\mathfrak{p}}(c)$ is a subset of $\bigcup_{i} \mathbf{S}[i]$. By the definition of leaf $\operatorname{supp}(c), \overline{\operatorname{Dom}(o)} \cap \operatorname{Dom}(c) \neq \emptyset$. By proposition 2.5, there is a point e in the closure set of o , $\mathrm{e} \in \operatorname{clos}(\mathrm{o})$, such that $\operatorname{Dom}(\mathrm{e}) \subseteq \operatorname{Dom}(\mathrm{c})$ or $\operatorname{Dom}(\mathrm{c}) \subseteq \operatorname{Dom}(\mathrm{e})$. By the definition of the global partition set $\mathcal{P}_{\Omega}$, the former must be true: otherwise, c could not be in $\overline{\mathcal{P}_{\mathfrak{p}}} \subseteq \mathcal{P}_{\Omega}$. Therefore o cannot be less refined than $c$, level $(\mathrm{o}) \geq \operatorname{level}(\mathrm{c})$. By proposition 2.6 there must be some support octant $s=\operatorname{supp}(c)[i]$ such that $o \in \operatorname{desc}(\mathbf{s})$. By the definition of $\mathbf{S}[i]$, it must contain o. This proves the first assertion.

From here, we split the proof into two cases, $\operatorname{dim}(c)=0$, and $\operatorname{dim}(c)>0$.
Suppose $\operatorname{dim}(c)=0$. If $o \in \operatorname{leaf} \operatorname{supp}_{\mathfrak{p}}(c)$, then there is $i$ such that $o \in \mathbf{S}[i]$. By proposition 2.8 , o must be an ancestor of the atom atom supp(c)[i]. Therefore $\circ$ is added to $\mathcal{L}$ on line 18 . Conversely, if $\circ$ is added to $\mathcal{L}$ on line 18 , then atomsupp $(\mathrm{c})[i]$ is a descendant of $o$, and by definition its domain is in o's domain, $\overline{\operatorname{Dom}(\operatorname{atom} \operatorname{supp}(c)[i])} \subseteq \overline{\operatorname{Dom}(o)}$. Because $\overline{\operatorname{Dom}(\operatorname{atom} \operatorname{supp}(c)[i])} \cap \operatorname{Dom}(c) \neq \emptyset$, it must be that $\overline{\operatorname{Dom}(o)} \cap \operatorname{Dom}(\mathrm{c}) \neq \emptyset$. Therefore o is a leaf in $\mathbf{S}[i] \subset \mathcal{O}_{\mathfrak{p}} \cup \mathbf{G}_{\mathfrak{p}}^{d}$ whose closure intersects c , which matches the definition of leaf $\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})$. Thus, if $\mathrm{c} \in \mathcal{P}_{\Omega}$, the set $\mathcal{L}$ computed is equal to leaf $\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})$, and the callback will be executed on line 20 if and only if $\mathrm{c} \in \overline{\mathcal{P}_{\mathfrak{p}}}$.

Now suppose $\operatorname{dim}(\mathrm{c})>0$. Let $L$ be the minimum level of a leaf o $\in \cup_{i} \mathbf{S}[i]$. The remainder of the proof is inductive on the difference $\delta=L-\operatorname{level}(\mathrm{c})$.

Suppose $\delta=0$, and let $o \in \mathbf{S}[i]$ be a leaf with level $L=\operatorname{level}(\mathrm{c})$. Because $o \subseteq \operatorname{supp}(c)[i]$ and because level $(\operatorname{supp}(c)[i])=\operatorname{level}(\mathrm{c})$ by definition, $o=\operatorname{supp}(c)[i]$.
 it must be that $\mathbf{S}[i]=\{\mathrm{o}\}$. Therefore $\circ$ is added to $\mathcal{L}$ on line 9 .

Because of the $2: 1$ condition, all remaining leaves in leaf $\operatorname{supp}_{\mathfrak{p}}(\mathrm{c})$ have level $L+1$. Let $o \in \mathbf{S}[j]$ be a leaf with level $L+1$. This implies that $\mathbf{S}[j] \neq\{\operatorname{supp}(c)[i]\}$, so the children of $\operatorname{supp}(c)[i]$ are assigned to $\mathbf{h}_{i}$ on line 12 o must be one of these children. On line 14 , o is added to $\mathcal{L}$ if and only if $\overline{\operatorname{Dom}(o)} \cap \operatorname{Dom}(c) \neq \emptyset$, which matches the Therefore, if $c \in \mathcal{P}_{\Omega}$, the constructed set $\mathcal{L}$ matches leaf $\operatorname{supp}_{\mathfrak{p}}(c)$, and the callback executes on line 20 if and only if $\mathrm{b} \in \overline{\mathcal{P}_{\mathfrak{p}}}$.

Now suppose the algorithm is correct for $0 \leq \delta<k$, and suppose $\delta=k$. There can be no $i$ such that $\mathbf{S}[i]=\{\operatorname{supp}(c)[i]\}$, so the arrays $\mathbf{H}_{i}$ and octants $\mathbf{h}_{i}$ are computed on lines 11 and 12 for every $i$. Let e be in the child partition set part(c): e has level level(c) +1 . By definition, each octant in the support set $\operatorname{supp}(e)$ also has level level $(\mathrm{c})+1$ and $\overline{\operatorname{Dom}(\operatorname{supp}(e)[i])} \cap \operatorname{Dom}(\mathrm{e}) \neq \emptyset$, which implies $\overline{\operatorname{Dom}(\operatorname{supp}(e)[i])} \cap$ $\operatorname{Dom}(\mathrm{c}) \neq \emptyset$. By proposition 2.6, implies there must be $j$ and $k$ such that $\operatorname{supp}(\mathrm{e})[i]=$ $\operatorname{child}(\operatorname{supp}(c)[j])[k]$. Therefore $\operatorname{supp}(\mathrm{e})[i]=\mathbf{h}_{j}[k]$ and the set $\mathbf{S}_{\mathrm{e}}[i]=\mathbf{H}_{j}[k]$ is equal to $\left(\mathcal{O}_{\mathfrak{p}} \cup \mathbf{G}_{\mathfrak{p}}^{d}\right) \cap \operatorname{desc}(\operatorname{supp}(\mathrm{e})[j])$. This means that the arguments of the recursive call on line 25 are correct for each $e \in \operatorname{part}(c)$. By the inductive assumption, the callback function is executed for the subset of $\overline{\mathcal{P}_{\mathfrak{p}}}$ whose domains are in $\operatorname{Dom}(c)=$ $\bigsqcup \operatorname{Dom}(\operatorname{part}(c)$. By the principle of induction, the proof is complete.

## Appendix C. Asymptotic analysis of Iterate (Algorithm 5.3).

We first present the asymptotic analysis of the complexity of the algorithm in a single-process, single-octree setting.

Theorem C.1. Ignoring the time taken by the callbacks, Iterate executes in the worst case in $O(N \log N)$ time.

Proof. The only operations in each instance of Iterate_interior that are not $O(1)$ are the $O(\log |\mathbf{S}[i]|)$ terms for the input arrays $\mathbf{S}[i]$. Each of these arrays is associated with an octant $\operatorname{supp}(c)[i]$ that is an ancestor of a leaf. An octant o can only be in $\operatorname{supp}(c)$ if $c \in \operatorname{bound}(o)$ and $c=(o, b)$ for some $b \in \mathcal{B}$. Therefore each ancestor octant can be associated with at most $|\mathcal{B}|$ terms with $O(\log |\mathbf{S}[i]|)$ complexity. An octree has $O(N)$ ancestors that are not leaves, so $O(N)$ searches are conducted. Each array $\mathbf{S}[i]$ contains a subset of leaves, so each $O(\log |\mathbf{S}[i]|)$ is $\mathcal{O}(\log N)$. We conclude that an upper bound on the running time is $O(N \log N)$.

Theorem C.2. Ignoring callbacks, Iterate executes in $O(N)$ time on a uniformly refined octree.

Proof. The leaves are all at the same level $L$, so $N=2^{d L}$, and there are $2^{d l}$ nodes in level $l$ of the tree. Because leaves are evenly distributed, each node at level $l$ has $2^{d(L-l)}$ leaf descendants. Each node is associated with a bounded number of binary searches and calls to Split_array, each with logarithmic complexity in the number
of leaves beneath it. So, ignoring leading coefficients, the time complexity is

$$
\begin{align*}
\sum_{l=0}^{L-1} 2^{d l} \log 2^{d(L-l)} & =d \sum_{l=0}^{L-1} 2^{d l}(L-l) \\
& =d \sum_{l=0}^{L-1} \frac{2^{d L}}{2^{d(L-l)}}(L-l)  \tag{C.1}\\
{[\hat{l}=L-l] } & =d 2^{d L} \sum_{\hat{l}=1}^{L} \frac{\hat{l}}{2^{d l}}=d 2^{d L} O(1)=d O(N) .
\end{align*}
$$

Because the dimension $d$ is fixed, Iterate runs in $O(N)$ time.
A uniformly refined octree is just a regular grid, so the indices of neighbors follow a predictable rule: a linear-time algorithm can be achieved without a recursive algorithm and without searching through the leaf arrays. We outline a class of octrees which has no rule for neighboring indices, but for which Iterate still runs in linear time.

Definition C. 3 ( $\Delta$-uniform octrees). A class of octrees is $\Delta$-uniform if the difference $\left(\max ^{\circ} \in \mathcal{O} 0 . l-\min _{0 \in \mathcal{O}}\right.$ o.l) is uniformly bounded by $\Delta$ for all octrees in the class.

Theorem C.4. Iterate executes in $O(N)$ time on a class of $\Delta$-uniform octree.
Proof. Let $L=\max _{0 \in \mathcal{O}}$ o.l and $l_{\min }=\min _{\mathrm{o} \in \mathcal{O}}$ o.l. For $l \min _{\mathrm{o} \in \mathcal{O}}$ o.l, $2^{d \ell}$ is now an upper bound on the number of nodes at level $l$, and for every $l, 2^{d(L-l)}$ is an upper bound on the number of descendant leaves of a level $l$ node. Therefore the $O\left(2^{d L}\right)$ runtime for a uniform octree is an upper bound on the runtime of Iterate, while a lower bound on $N$ is $2^{d l_{\text {min }}}=2^{d(L-\Delta)}$. Therefore $2^{d L} \leq 2^{d \Delta} N$, so the runtime of Iterate is $O\left(2^{d \Delta} N\right)=O(N)$.

We now consider the Iterate algorithm in the multiple process, single octree setting, and derive bounds in terms of the local number of leaves $N_{\mathfrak{p}}$ and the number of processes $P$. A key component of the above analysis for the serial runtime, that the number of ancestor nodes is $O(N)$, is no longer true in a parallel setting: the number of ancestors of the leaves in $\mathcal{O}_{\mathfrak{p}} \cup \mathbf{G}_{\mathfrak{p}}^{d}$ is not necessarily $O\left(N_{\mathfrak{p}}\right)$. Suppose a is the smallest common ancestor of every leaf in $\mathcal{O}_{\mathfrak{p}} \cup \mathbf{G}_{\mathfrak{p}}^{d}$ and a.l $=\hat{l}$. The number of branches below a must be $O\left(N_{\mathfrak{p}}\right)$, so the analysis for the runtime after level $\hat{l}$ is the same as for a single process, substituting a for the root, so the time spent below a is $O\left(N_{\mathfrak{p}} \log N_{\mathfrak{p}}\right)$ in general or $O\left(N_{\mathfrak{p}}\right)$ for a $\Delta$-uniform tree. Thus an upper bound for the runtime is to add $O\left(\hat{l} \log N_{\mathfrak{p}}\right)$ to that time. We can bound $\hat{l}$ by $L=\max _{\mathrm{o} \in \mathcal{O}}$ o.l, and in the $\Delta$-uniform case $L \in O(\log N)$. If we assume an even partitioning of the leaves, $N=P N_{\mathfrak{p}}$, then $L \in O\left(\log P+\log N_{\mathfrak{p}}\right)$. The runtime for Iterate on an evenly distributed octree is thus $O\left(\left(L+N_{\mathfrak{p}}\right) \log N_{\mathfrak{p}}\right)$ in general and $O\left(\log P+N_{\mathfrak{p}}\right)$ for $\Delta$-uniform octrees.

Introducing multiple trees does not affect the analysis significantly: maintaining separate arrays for each tree can only reduce the sizes of the subarrays that are split by Split_array. Some time is taken to set up the calls to Iterate_interior for the interfaces between octrees, but this time is negligble, especially if the forest realizes the common use case $K \ll N$.


[^0]:    ＊Institute for Computational Engineering and Sciences，The University of Texas at Austin，USA
    ${ }^{\dagger}$ Institut für Numerische Simulation，Rheinische Friedrich－Wilhelms－Universität Bonn，Germany
    $\ddagger$ Department of Applied Mathematics，Naval Postgraduate School，USA
    §Jackson School of Geosciences，The University of Texas at Austin，USA
    『Department of Mechanical Engineering，The University of Texas at Austin，USA

[^1]:    ${ }^{1}$ This data type is a notational convenience for this work, not part of the p4est interface.

[^2]:    ${ }^{2}$ In p4est, the tree index is always available from context, not stored with the octant.

[^3]:    ${ }^{3}$ By "geometrically conformal mesh" we mean that $\left\{T^{t}\right\}_{0 \leq t<K}$ are the cells of a CW complex (see e.g. 19. Chapter 10]): informally, each $T^{t}$ is open, $\left\{T^{t}\right\}_{0 \leq t<K}$ tiles $\Omega$, and if the intersection $\overline{T^{s}} \cap \overline{T^{t}}$ has dimension $(d-1)$, then it is equal to a whole face of $T^{s}$ and a whole face of $T^{t}$.

[^4]:    ${ }^{4}$ In p4est partitions may be empty, but for simplicity we assume here that they are all non-empty.

[^5]:    ${ }^{5}$ In p4est, the implementation differs: rather than the per-node lists g. $\mathcal{S}_{\text {share }}$, we store lists of nodes that are shared by each process, which is a more useful layout for filling communication buffers.

[^6]:    ${ }^{6}$ https://perf.wiki.kernel.org

[^7]:    ${ }^{7}$ The number of global nodes depends on the forest topology and the refinement pattern. For a single octree with uniform refinement, the number of global nodes is asymptotically equivalent to $n^{3} N$, in which case the number of 7 th-order nodes would be 343 times the number of 1st-order nodes. Because of non-conformal elements, however, we see a higher ratio.

