Provided for non-commercial research and educational use only.
Not for reproduction or distribution or commercial use.

Volume 67 Number 4 April 2007 ISSN 0743-7315

Journal of Parallel and Distributed Computing

EDITORS
Allan Gottlieb, New York University
Kai Hwang, University of Southern California
Sartaj Sahni, University of Florida

- *Theory of Parallel/Distributed Computing*
- *Innovative Computer Architectures*
- *Supercomputers and Numerical Methods*
- *Software Tools and Environments*
- *Parallel Algorithms and Implementations*
- *Languages, Compilers, and Operating Systems*
- *Shared Memory Multiprocessors*
- *Multicomputers and Distributed Processing*
- *Fault–Tolerant Computing*
- *Artificial Intelligence Machines*
- *Applications and Performance Analysis*
- *Optical Computing*
- *Neural Networks*

Available online at
ScienceDirect
www.sciencedirect.com

This article was originally published in a journal published by Elsevier, and the attached copy is provided by Elsevier for the author's benefit and for the benefit of the author's institution, for non-commercial research and educational use including without limitation use in instruction at your institution, sending it to specific colleagues that you know, and providing a copy to your institution's administrator.

All other uses, reproduction and distribution, including without limitation commercial reprints, selling or licensing copies or access, or posting on open internet sites, your personal or institution's website or repository, are prohibited. For exceptions, permission may be sought for such use through Elsevier's permissions site at:

http://www.elsevier.com/locate/permissionusematerial

Available online at www.sciencedirect.com
ScienceDirect

ELSEVIER

J. Parallel Distrib. Comput. 67 (2007) 417–429

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Author's personal copy

A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids[☆]

William F. Mitchell

Mathematical and Computational Sciences Division, National Institute of Standards and Technology, Gaithersburg, MD 20899-8910, USA

Received 21 December 2005; received in revised form 7 July 2006; accepted 28 November 2006
Available online 7 February 2007

Abstract

The partitioning of an adaptive grid for distribution over parallel processors is considered in the context of adaptive multilevel methods for solving partial differential equations. A partitioning method based on the refinement-tree is presented. This method applies to most types of grids in two and three dimensions. For triangular and tetrahedral grids, it is guaranteed to produce connected partitions; no other partitioning method makes this guarantee. The method is related to the OCTREE method and space filling curves. Numerical results comparing it with several popular partitioning methods show that it computes partitions in an amount of time similar to fast load balancing methods like recursive coordinate bisection, and with mesh quality similar to slower, more optimal methods like the multilevel diffusive method in ParMETIS.
Published by Elsevier Inc.

Keywords: Adaptive refinement; Dynamic load balancing; Grid partitioning; Multilevel adaptive method; Parallel finite elements; Refinement tree; Space filling curves

1. Introduction

An adaptive multigrid method solves an elliptic partial differential equation (PDE) by beginning with a very coarse grid and cycling through phases of adaptive refinement/derefinement of the grid and multigrid solution of the linear system of equations resulting from discretization of the PDE on the adaptive grid. In a parallel adaptive multigrid method, the adaptive refinement phase can cause the load balance over the processors to become unequal. If the load is too unbalanced, the grid must be repartitioned and redistributed before continuing with the solution phase.

An important part of a parallel adaptive multigrid method is the method for determining this partition. In this context, it must not only produce equal sized sets to balance the load and minimize cut edges to reduce communication, but must also be very fast to not dominate the computation time of a fast multigrid method, and must produce similar partitions on a grid and a refinement/derefinement of that grid to reduce redistribution costs.

In this paper we present the refinement-tree partitioning method (REFTREE), a new method for partitioning grids that were created by adaptive refinement. The method was developed as a k-way version of the recursive bisection refinement-tree method [7] to reduce the amount of communication overhead in a parallel implementation. It is also very similar to the Octree partition method (OCTREE) [4] and is related to space filling curve (SFC) methods [3,11,12,15]. The primary difference between REFTREE and OCTREE is the generation of the tree. In OCTREE the tree represents a geometric refinement of a region covering the domain through local subdivision of octants (or quadrants in two dimensions), while in REFTREE the tree represents the refinement of some initial set of coarse grid elements. The primary difference between REFTREE and the traditional SFC methods is one of geometric vs. algebraic orientation. In both algorithms, a SFC is used to create a linear sequence of the elements. The sequence is then cut into segments, which become the partitions of the grid. SFC is more

[☆] This research was partially supported by Sandia National Laboratories under contract AR-1283. Contribution of NIST, not subject to copyright.

E-mail address: william.mitchell@nist.gov
URL: http://math.nist.gov/∼WMitchell.

0743-7315/$ - see front matter Published by Elsevier Inc.
doi:10.1016/j.jpdc.2006.11.003

algebraic in that it performs computations on the coordinates of the centroids of the elements to determine a SFC ordering. In contrast, REFTREE uses geometric subdivision of an initial set of elements to determine a SFC ordering, which is more flexible for complicated geometries. Neither OCTREE nor SFC can guarantee that the partitions will be connected. REFTREE is guaranteed to produce connected partitions for triangular and tetrahedral grids, and we have observed that it usually does for other types of grids.

The rest of the paper is organized as follows. Section 2 defines what is meant by the refinement-tree of an adaptive grid that was created by local refinement of an initial coarse grid. Section 3 then describes how to order the nodes in the refinement-tree such that a depth-first traversal of the tree induces a SFC through the grid. Section 4 presents the refinement-tree partitioning algorithm in both sequential and parallel form. Section 5 contains numerical results comparing REFTREE to several other partitioning methods.

2. Refinement-tree

In this section we define the refinement-tree of a locally refined grid. The refinement-tree is a representation of the refinement process that created the grid. It contains all the information about how an element was created by refinement, but does not indicate the order in which elements were refined.

Let Ω be a closed, connected, bounded region in R^d, $d = 2, 3$. For simplicity we assume that Ω is polygonal, but that is not necessary if one uses elements with curved edges (faces), or allows the grid to approximate Ω. We define a *grid* on Ω, $G = \{E_i\}_{i=1}^N$ to be a set of elements, E_i, such that E_i is a polygon in R^d, $\breve{E}_i \cap \breve{E}_j = \emptyset$, $i \neq j$, and $\cup E_i = \Omega$, where $\breve{E}_i$ denotes the interior of E_i and $\emptyset$ is the empty set. If, in addition, the intersection of any two elements is either empty, a common vertex, a common side, or, in R^3, a common face, then the grid is said to be *conforming*. Typically in R^2 an element is a triangle or quadrilateral, and in R^3 an element is a tetrahedron or hexahedron, although other elements (for example, prisms) are sometimes used. In practice one only uses conforming triangular and tetrahedral grids, but allows quadrilateral and hexahedral grids to be non-conforming (also referred to as "having hanging nodes").

A *locally refined grid* is obtained from an initial grid G_0 by subdividing (or *refining*) some of the elements, and possibly further subdividing some of the resulting elements, etc. An element is normally refined into elements of the same type (e.g. triangles are refined into triangles). Elements can also be derefined by joining the resulting elements back together to form the original element. Fig. 1 illustrates the most commonly used methods of refining elements.

The *refinement-tree* of a locally refined grid, $T(G) = \{V, \{C(v_i)\}\}$ consists of a set of nodes, $V = \{v_i\}_{i=0}^M$, and for each $v_i \in V$ a set of children $C(v_i) \subset V$. Each node $v_j \in V$ is contained in exactly one set $C(v_i)$, except for v_0 which is called the *root* and not contained in any $C(v_i)$. If $v_j \in C(v_i)$ then v_j is a *child* of v_i and v_i is the *parent* of v_j. If $C(v_i) = \emptyset$ then v_i is called a *leaf*. An *ancestor* of v_i is any node on the (unique) path between v_i and the root. The *descendants* of v_i are the nodes in the subtree rooted at v_i. When depicted graphically the nodes are drawn as circles and the children are drawn below their parent and connected to the parent with an edge.

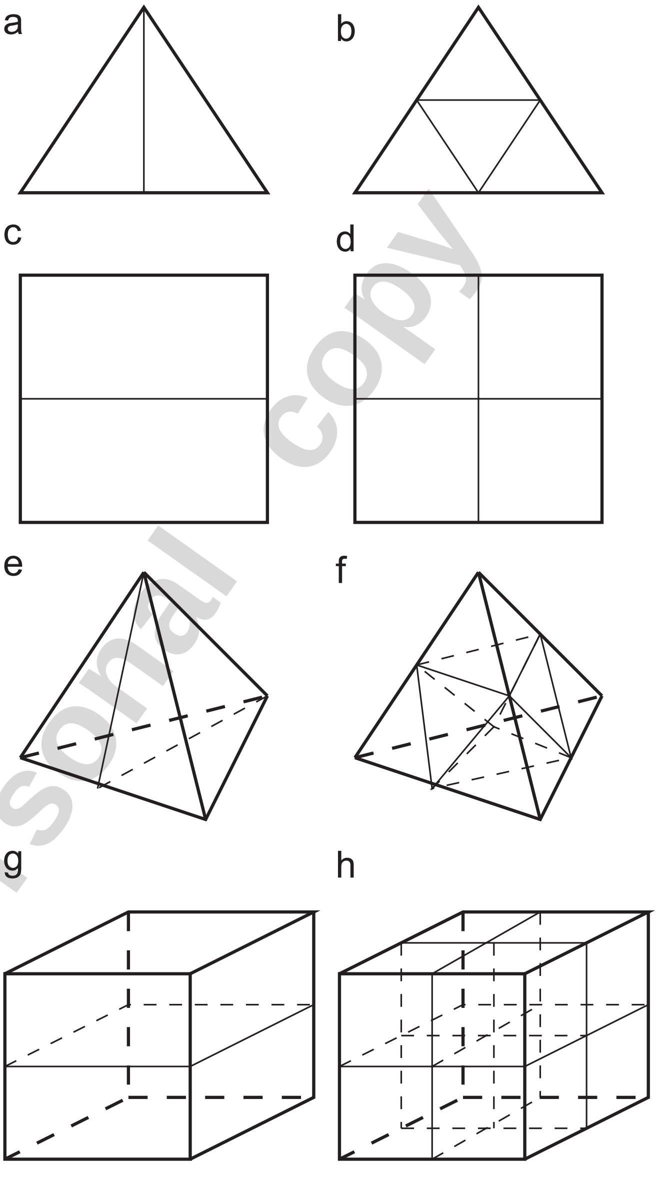

Fig. 1. Element refinements: (a) triangle bisection; (b) triangle quadrisection; (c) quadrilateral bisection; (d) quadrilateral quadrisection; (e) tetrahedron bisection; (f) tetrahedron octasection; (g) hexahedron bisection; (h) hexahedron octasection.

The refinement-tree corresponds to the grid as follows. The root corresponds to Ω. The children of the root are in one-to-one correspondence with the elements in the initial grid G_0. The children of any other node correspond to the elements that were created when the corresponding element was refined. Note that the leaves correspond to the elements of the grid G. Fig. 2 illustrates the correspondence between a locally refined grid that was created by bisection of triangles and its refinement-tree. The left side of the figure shows the sequence of refinements

Author's personal copy

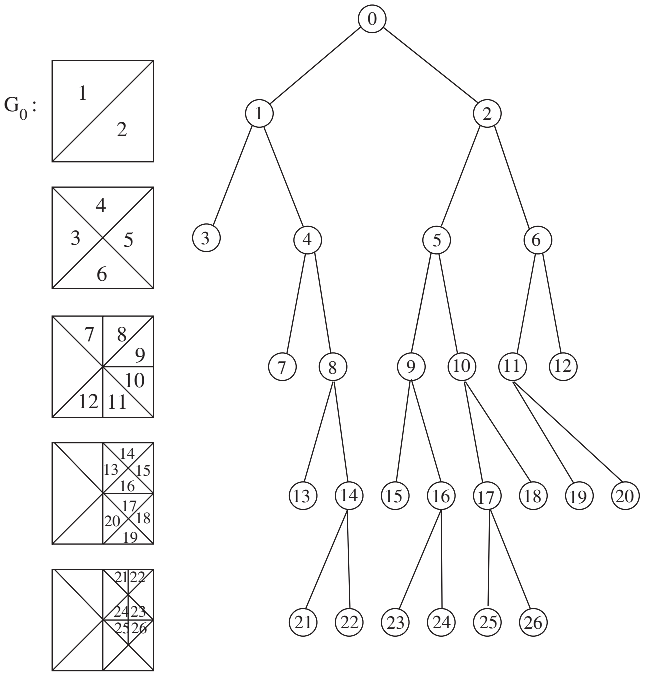

Fig. 2. Correspondence between grid refinement and the refinement-tree.

that created the grid, and the right side shows the refinement tree. The numbers show the correspondence between elements of the grid and nodes of the tree. Unlabeled elements have the same number as in the grid above them. We will often refer to the element or the corresponding node interchangeably.

In a distributed-memory parallel application the grid will be distributed across the processors of a parallel computer. Consequently, the refinement-tree will also be distributed. Fig. 3 illustrates the distribution of the refinement-tree of Fig. 2 over two processors. The elements of the grid are drawn in red and green to show the two partitions which are assigned to processors 1 and 2, respectively. The two trees show the *local refinement-tree* for each processor, which represent the part of the grid assigned to that processor. Clearly the local refinement-tree must contain the leaves assigned to the processor (to represent the partition), and the ancestors of those leaves (to be a tree). These nodes are colored red in the local refinement-tree for processor 1 and green in the local refinement-tree for processor 2. There is overlap between the local refinement-trees, since some parents will have their children assigned to different processors, and each of those processors must then contain the parent in its local refinement-tree. The parallel REFTREE algorithm of Section 4 requires slightly more overlap. It also requires that any non-leaf node of the local refinement-tree has *all* of its children of the global refinement-tree present. These nodes are colored white in Fig. 3. Note that this requires all processors to have a node corresponding to every element of the initial grid, since they are all children of the root. This does not, however, imply that every processor needs the full data structure for every element of the initial grid. A "light weight" substitute data structure can be used for initial elements assigned to other processors. This defines a minimal local refinement-tree; additional nodes can be included if desired. For example, one may want to include additional nodes so that the local refinement-tree corresponds to a conforming grid.

3. Order of children

In the refinement-tree partitioning algorithm defined in Section 4, we perform depth-first traversals of the refinement-tree. In order for the algorithm to produce connected partitions it is crucial that the traversal visit the children of a node in the correct order. Thus $C(v_i) = \{c_1(v_i), c_2(v_i), \ldots, c_{m_i}(v_i)\}$ is an ordered set of children. Here $m_i = |C(v_i)|$ is the number of children of node v_i.

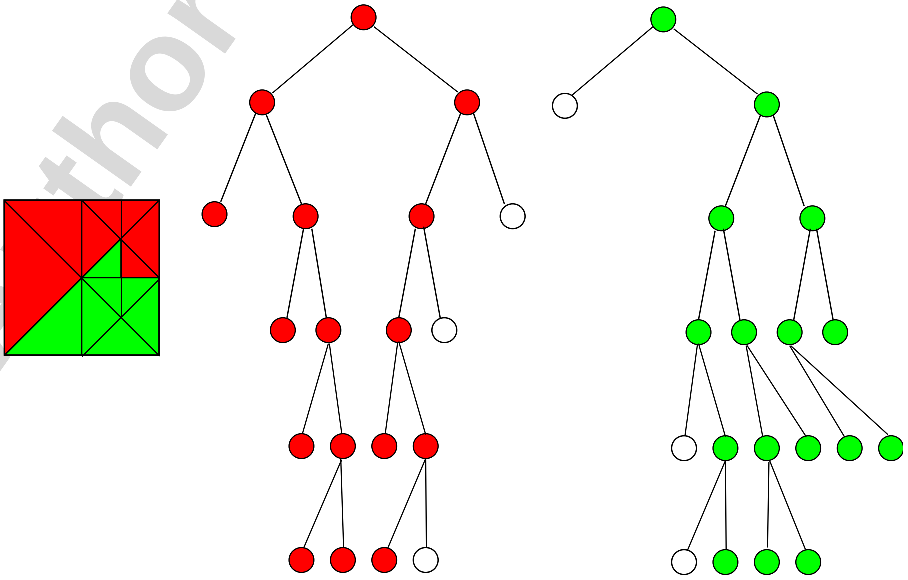

Fig. 3. A refinement-tree distributed over two processors.

Author's personal copy

Authors personal copy

3.1. In-vertex and out-vertex

To facilitate the definition of the order of the children of a node we designate two special vertices of an element as the *in-vertex* and *out-vertex*. The in-vertex must not be the same as the out-vertex.

In a depth-first traversal of the tree, the leaves of the refinement-tree are visited in some order, or equivalently the elements of G are visited in some order. The goal of the ordering of the children is to create a refinement-tree for which the traversal will visit the elements in an order where each element is connected to the preceding element. In other words, with this ordering of the elements we can draw a connected curve that passes through each element of G exactly once. If the grid is conforming then two connected elements must share a common vertex, and the curve can be drawn through the vertex, which can be designated as the out-vertex of the element that is visited first and the in-vertex of the other element. Such a sequence of connected elements and in/out-vertices is called a *through-vertex Hamiltonian path* of G.

3.2. Elements of the initial grid

Consider first the ordering of the children of the root, i.e. the elements of the initial grid G_0. The following theorem is proven in [10].

Theorem 1. *Let G be a conforming triangular or tetrahedral grid with at least two elements. If G contains no local cut vertices and, for tetrahedra, no local cut edges, and contains at least one interior vertex, then there exists a through-vertex Hamiltonian path for G.*

A local cut vertex is a point which, if removed, would cause the grid to become disconnected locally. In other words, given any two elements that contain that vertex, their intersection is only that vertex. A local cut edge is defined similarly. An interior vertex is a vertex that is not on the boundary of Ω. For tetrahedral grids, the existence of an interior vertex is not necessary.

It was also shown in [10] that it is not always possible to find a path that passes through the element sides instead of vertices. This implies that it is not always possible to find a partitioning of the elements where every element shares a side with another element in the same partition. However, Theorem 1 implies that we can always find connected partitions, even if only connected by a single vertex.

Unfortunately, the same result does not hold for quadrilaterals and hexahedra. In fact, it is easy to construct examples for which no through-vertex Hamiltonian path exists. However, such a path often exists, and in most cases when it does not, one can find a sequence of elements in which there are only a few “discontinuities” where adjacent elements in the path are not connected.

The proof of Theorem 1 in [10] is a constructive proof which leads to an efficient algorithm for constructing a through-vertex Hamiltonian path for triangular and tetrahedral grids. An algorithm is also given for quadrilaterals and hexahedra. We do not repeat those algorithms here, but refer to [10] for the details. The results of these algorithms then provide the order of the children of the root, and the assignment of the in-vertex and out-vertex of the elements of G_0. We comment that the algorithms in [10] will produce a through-vertex Hamiltonian path for G_0, but tend to produce paths that are not compact, and hence partitions with large boundaries. Therefore, some other algorithm may be preferred for processing the initial grid G_0.

3.3. Elements created by refinement

Next, consider the ordering of the children of the non-root nodes. Here we seek a through-vertex Hamiltonian path through the children that begins at the in-vertex of the parent and ends at the out-vertex of the parent. This insures that if we have a through-vertex Hamiltonian path in the grid before refining the parent, then we still have one after refinement. In fact, the segment of the path that passes through the parent is replaced by a segment that passes through all the children in a continuous manner.

In general, one can find a path through the children, if one exists, by exhaustive search. Although the complexity of exhaustive search is exponential in the number of children, a refinement strategy typically produces a small number of children so it is not prohibitive. On the other hand, it is more efficient to use a set of templates for a given refinement strategy because the number of cases that arise is small.

The templates for triangle bisection are given in Fig. 4. There are three cases depending on the relationship between the vertex opposite the side to be bisected and the in-vertex and out-vertex. Fig. 4(a) shows the case where the vertex opposite the side to be bisected is neither the in-vertex nor the out-vertex. The parent’s in-vertex and out-vertex are labeled “I” and “O”, respectively, outside the parent element. The children’s in-vertices and out-vertices are labeled inside the child elements. To identify the order of the children, begin at the in-vertex of the parent and traverse the children going through the out-vertex and in-vertex of the next child. Fig. 4(b) shows the case where the out-vertex is opposite the side to be bisected, and Fig. 4(c) shows the case where the in-vertex is opposite the side to be bisected.

The templates for five of the other refinement strategies of Fig. 1 are shown in Figs. 5–9. For triangle quadrisection and tetrahedra octasection, there is only one template. There are two cases for quadrilateral quadrisection, depending on whether the out-vertex is adjacent to the in-vertex or is the opposite corner. There are four cases for tetrahedra bisection, depending on whether or not each of the in-vertex and out-vertex is on the edge to be bisected. And there are three cases for hexahedron octasection, depending on whether the out-vertex shares an edge with the in-vertex, shares a face but not an edge, or is the opposite corner.

The assignment of in- and out-vertices to children does not work for all refinement strategies. In particular it can fail for a refinement strategy that produces a child that contains two vertices that are not contained in any other child. If those vertices

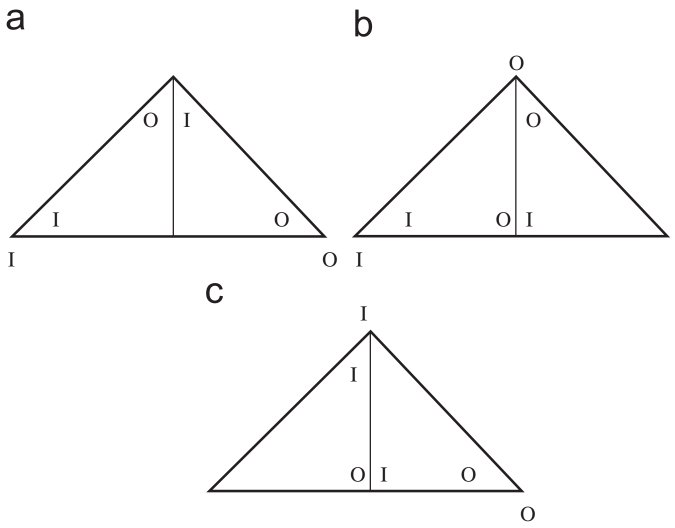

Fig. 4. The three templates for child order and in/out-vertices for refinement by triangle bisection.

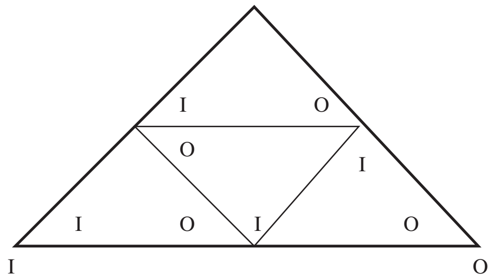

Fig. 5. The template for child order and in/out-vertices for refinement by triangle quadrisection.

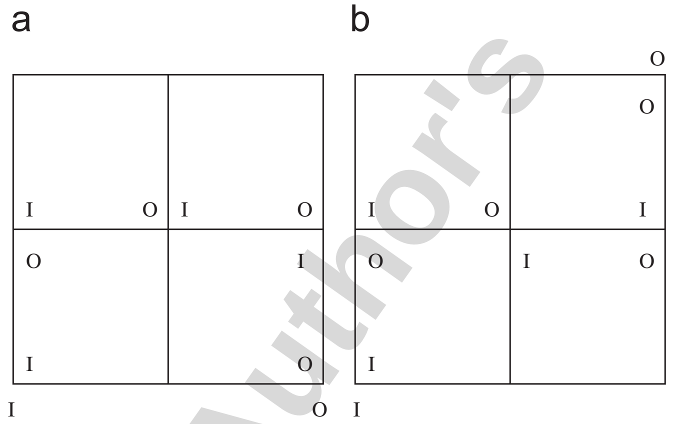

Fig. 6. The two templates for child order and in/out-vertices for refinement by quadrilateral quadrisection.

happen to be the in-vertex and out-vertex of the parent, then it will be impossible to start at the in-vertex, pass through all the children, and end at the out-vertex without visiting the same element twice. This is illustrated in Fig. 10 for bisected quadrilaterals. It also occurs with bisected hexahedra. These are the only two commonly used refinement strategies for which the method fails, as far as the author knows.

Within an element, the traversal of the descendants is in a hierarchical manner, and does not leave an element until all the descendants have been visited. This leads to a SFC within each of the initial elements. Many of these are well-known SFCs [13]. For example, quadrilateral quadrisection leads to the Hilbert SFC (HSFC) (Fig. 11), and triangle bisection leads to the Sierpinski SFC (Fig.12).

The connectedness of the SFC within each element, together with Theorem 1 and the connectedness of a through-vertex Hamiltonian path, lead to the following result.

Theorem 2. *Let a partition be the set of elements contained in any segment of the ordered list of elements obtained by a depth-first traversal of a refinement-tree with the children ordered by the methods described in this section.*

(1) *If the grid is a triangular grid refined by bisection or quadrisection, or a tetrahedral grid refined by bisection or octasection, then the partition is connected.*
(2) *If the grid is a quadrilateral grid refined by quadrisection or a hexahedral grid refined by octasection, and a through-vertex Hamiltonian path is found for the initial grid, then the partition is connected.*

4. Refinement-tree partitioning method

This section describes the refinement-tree (REFTREE) algorithm. We first present it as a sequential algorithm, and then present the parallel form.

In REFTREE, the nodes of the refinement-tree are weighted. The weight assigned to a node should be related to the amount of work associated with the corresponding element. For example, elements containing a Dirichlet boundary may have a smaller weight than elements in the interior of the domain. The weights are not limited to leaf nodes; one may wish to apply weights to interior nodes to, for example, represent the work on a coarse grid of a multigrid solver. In the simplest weighting scheme, the leaf nodes have weight 1.0 and the interior nodes have weight 0.0. This results in a partitioning of the elements of the final grid into equal sized sets (or differing by at most one if the number of elements is not a multiple of the number of partitions). For predictive load balancing (where the new partition is computed before the grid is refined), the error indicator (used by an adaptive refinement method to determine which elements get refined [6]) is a reasonable weight, since one expects the elements with larger error indicators to be refined more times.

REFTREE consists of two phases. In the first phase, every node is labeled with the sum of the weights in the subtree rooted at that node. This is accomplished by a depth-first traversal of the tree, and takes $O(N)$ operations. In the second phase, a truncated depth-first traversal of the tree is performed to create the partitions. The desired size of each partition is determined by dividing the summed weight at the root node by the number of partitions, and the partitions are initialized to be empty. During the traversal, the summed weights at the nodes are examined relative to the size of the partition under

Author's personal copy

Author's personal copy

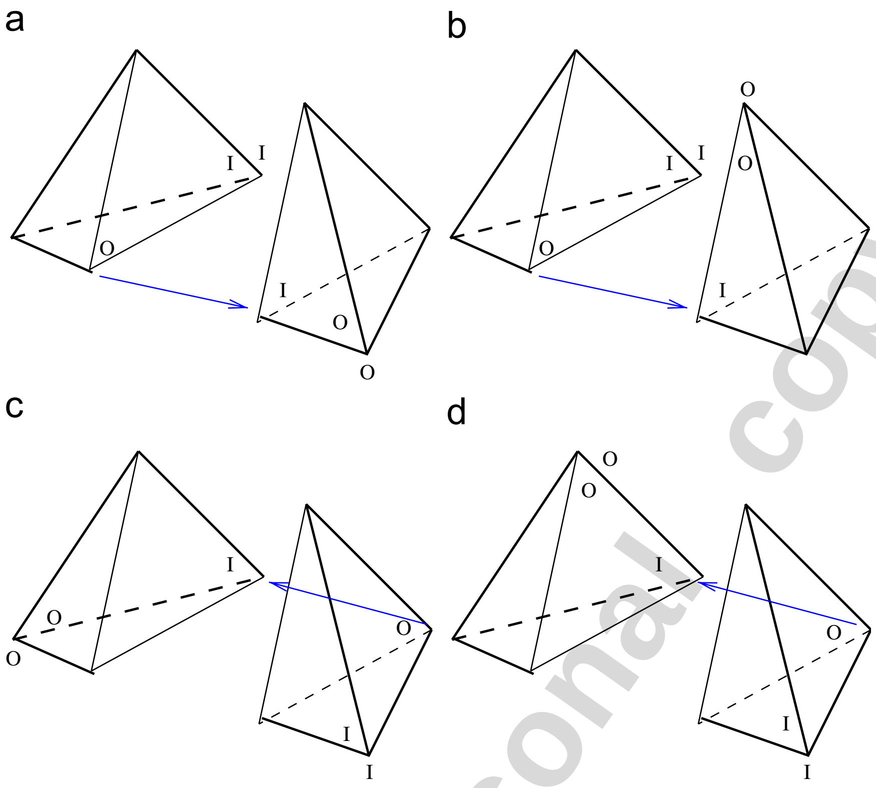

Fig. 7. The four templates for child order and in/out-vertices for refinement by tetrahedra bisection.

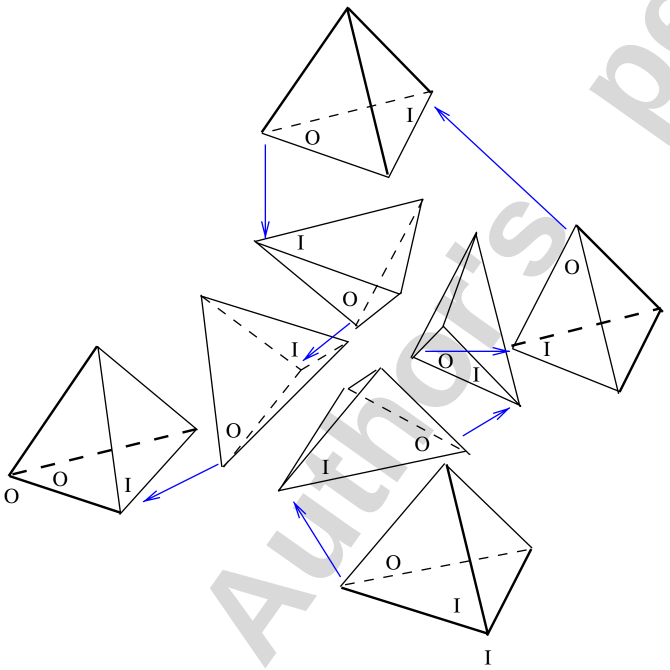

Fig. 8. The template for child order and in/out-vertices for refinement by tetrahedra octasection.

construction. If it is small enough to be added to the partition without exceeding the desired size, then it is added and the subtree is not traversed. Otherwise, the children are visited and the subtree will be split among two or more partitions. Fig. 13 illustrates this process for partitioning the grid of Fig. 2 into two parts with equal number of elements. The leaves have weight 1 and the interior nodes have weight 0. The color of the elements in the sequence of grids shows the sets of elements that were added to a partition in each step. The numbers in the nodes of the refinement-tree are the summed weights. Fig. 14 gives the sequential algorithm, using the notation of Section 2.

Most of the time the subtree will fit in the current partition, so large portions of the grid are assigned to a partition at the same time. Only when a partition is nearly full will the traversal go deep into the tree to find a small enough subtree to fill out the partition. If there are p partitions and the depth of the tree is $O(\log N)$, one would expect the second phase to require $O(p \log N)$ operations.

In a parallel implementation, the refinement-tree is distributed over the p processors, as described in Section 2. Since the refinement-tree is distributed, the summation of the weights must be done in a distributed manner. This can be accomplished with two tree traversals and one all-to-all communication step. In the first tree traversal, the weights are summed for nodes that belong to this processor. Leaf nodes of the local refinement-tree that are not leaf nodes of the global refinement-tree, called *pruning points*, are given the weight 0.0, but otherwise the summation occurs as usual. The processors then exchange information to provide the summed weights for the pruning points, by each processor sending what it has as the summed weight for each node that is a pruning point on a different processor. Note that a processor may receive contri-

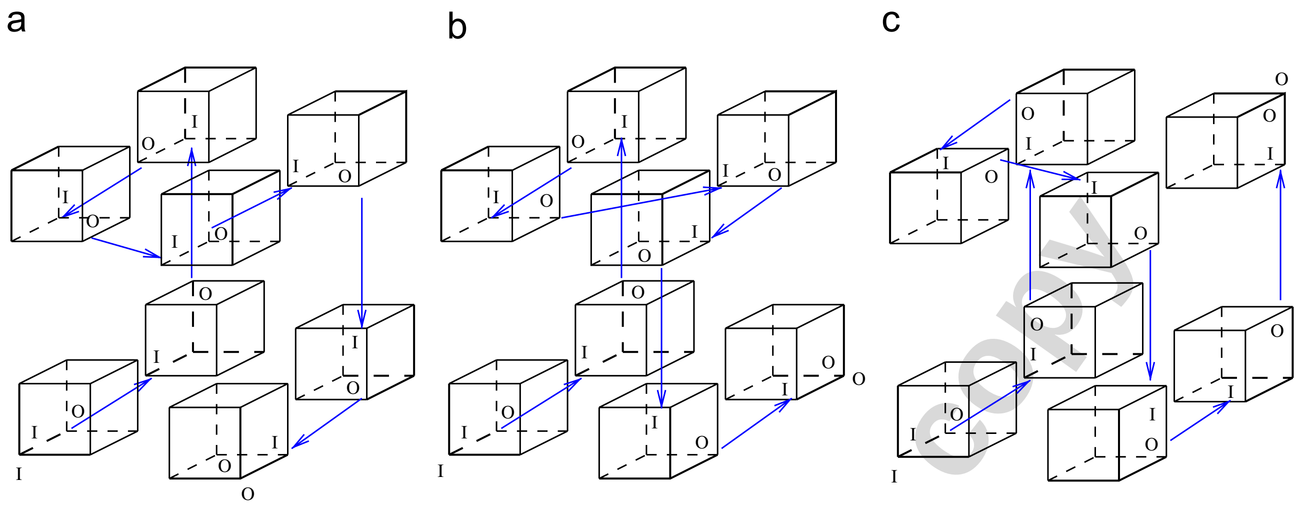

Fig. 9. The three templates for child order and in/out-vertices for refinement by hexahedra octasection.

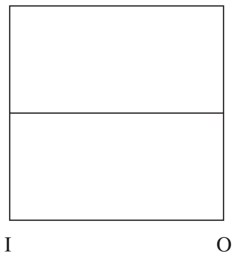

Fig. 10. Example that illustrates that the method can fail for bisected quadrilaterals.

butions for a pruning point from more than one processor, and the sum of these is the correct summed weight for that node. Now with the summed weight available for the pruning points, a second traversal is performed to obtain the correct summed weights for the entire tree.

Each processor now has sufficient information to perform the second phase of the REFTREE algorithm independently. Without further communication, all processors will obtain the same partition, except for details within parts of the grid that the processor does not have. These details are not needed since all the processor needs to know is the new assignment for the elements it currently has so that it can send those elements to the new owner during the redistribution of data.

For a grid with N elements, if each processor has $O(N/p)$ elements and $O((N/p)^r)$, $r \leqslant 1$, "shadow" (or "ghost") elements, the expected number of operations on each processor is $O(N/p)$. Typically $r = \frac{1}{2}$ for two-dimensional grids and $r = \frac{2}{3}$ for three-dimensional grids.

5. Numerical results

Numerical experiments were performed to demonstrate the performance of the refinement-tree partitioning method and to compare it to several other partitioning methods for dynamic load balancing. Three example problems were used: (1) a problem that is commonly used for adaptive refinement demonstra-

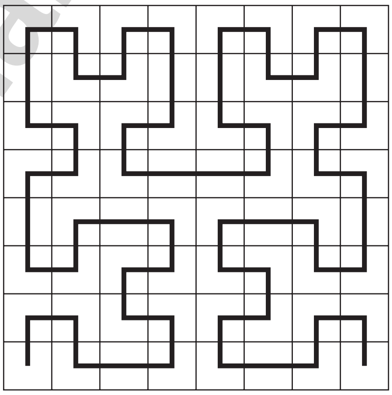

Fig. 11. The Hilbert space filling curve.

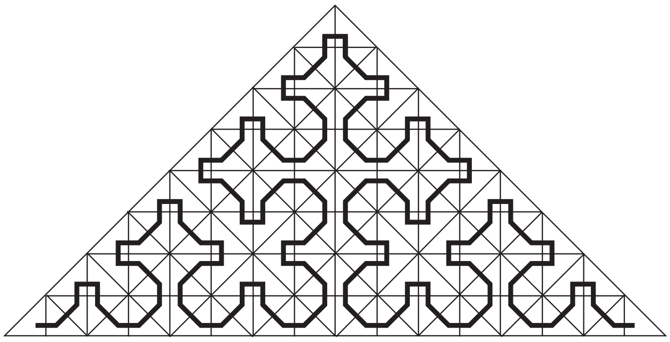

Fig. 12. The Sierpinski space filling curve.

tions; (2) a two-dimensional problem on a more complicated domain; and (3) a three-dimensional problem.

The computations for the two-dimensional problems were performed on a cluster of 1.66 GHz AMD Athlon MP 2000[1]

[1] The mention of specific products, trademarks, or brand names is for purposes of identification only. Such mention is not to be interpreted in any way as an endorsement or certification of such products or brands by the National Institute of Standards and Technology. All trademarks mentioned herein belong to their respective owners.

Author's personal copy

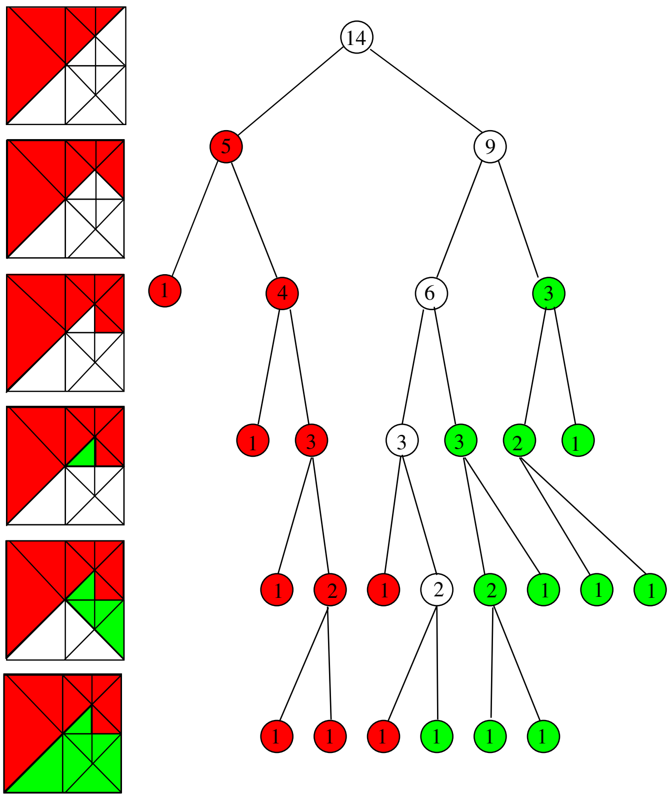

Fig. 13. Partitioning the grid and refinement-tree into two sets.

based PCs operating under the Red Hat 7.3 distribution of Linux with kernel 2.2.20. Programs were compiled with Lahey Fortran 95 version 6.1 and gcc 2.96. Message passing was performed with the LAM 6.5.9 implementation of MPI. The computations for the three-dimensional problem were performed on a cluster of 2.4 GHz Xeon based PCs with Fedora Core 3, Linux kernel 2.6.10, Lahey Fortran 95 version 6.2, gcc 3.4.3, and LAM 7.1.1.

The two-dimensional problems were solved with PHAML version 0.9.13 [8,9]. This is an adaptive multilevel program that uses triangle bisection for refinement. The grid for the three-dimensional problem was generated by hexahedron octasection outside the context of a PDE solver. All partitions were computed by Zoltan version 1.52 [1,2], which includes ParMETIS version 3.1 [5,14].

The partitioning methods used in these computations are:

(1) refinement-tree (REFTREE)—the method described in this paper;
(2) Hilbert space filling curve (HSFC)—a traditional SFC method;
(3) OCTREE (OCTREE)—the OCTREE method;
(4) ParMETIS (ParMETIS)—a multilevel diffusive method (ParMETIS_RepartLDiffusion) from the popular static partitioning library;
(5) recursive coordinate bisection (RCB)—a method that recursively bisects the region into two parts, with separators parallel to the axes; and
(6) recursive inertial bisection (RIB)—a method that recursively bisects the region into two parts, with separators orthogonal to the longest direction of the subregion.

Further details of the methods can be found in [2] and the references therein. We comment on the expected performance of two of the methods. First, this implementation of the OCTREE method was intended for three-dimensional grids. It partitions a two-dimensional grid by embedding it in three dimensions and partitioning it as a three-dimensional grid. Consequently we expect OCTREE to have a longer computation time than would be possible with a two-dimensional implementation (hence the timing results are not reliable), but to give the expected quality of partition (hence the quality results may be compared with the other methods). Second, ParMETIS was intended to be a near-optimal static partitioner for a large number of processors. Consequently we expect ParMETIS to have longer computation times but produce higher quality partitions.

The first example is a problem that is often used to demonstrate adaptive refinement. It is Laplace's equation with Dirichlet boundary conditions on an L shaped domain. There is a singularity at the reentrant corner, which induces smaller elements in that region. An adaptive grid for this problem is shown in Fig. 15. Sample partitions for 16 processors are shown in Fig. 16 for each of the methods.

The program begins with a grid containing six triangles and refines it to approximately 16,000 nodes sequentially. It then cycles through three phases: (1) partition the grid, (2) refine to approximately double the number of nodes, and (3) solve the linear system using two multigrid V-cycles. The program terminates when there are approximately one million nodes. Runs were made using from 2 to 32 processors (the number of partitions equals the number of processors). Since partitioning occurs before refinement, this is a predictive load balancing approach. The weights on the elements are the error indicators used by the adaptive refinement algorithm, which in this case is a simple hierarchical coefficient estimator [6]. The wall clock time for each partitioning phase was measured using the Fortran subroutine system_clock. The number of elements moved between processors was counted at the end of each partitioning phase. The number of cut edges in the dual graph of the grid (i.e., the number of adjacent elements assigned to different partitions) was counted after each refinement phase.

Fig. 17 shows the time for computing each partition by REFTREE for 2, 4, 8, 16 and 32 processors. The $O(N)$ growth is clearly demonstrated independent of the number of processors. $O(1/p)$ growth is observed for 2, 4 and 8 processors, but the method slows down for 16 and 32 processors. This is primarily due to increasing communication costs in the all-to-all communication of the algorithm, which is not reflected in the operation count of Section 4. Fig. 18 shows the execution time of REFTREE with the communication time removed. Here we observe the decrease in time as the number of processors is increased. This indicates that the REFTREE method with all-to-all communication may not be appropriate for large numbers of

Author's personal copy

```
procedure REFTREE
  sum_weights($v_0$)
  make_partitions($v_0$,1,0,summed_weight($v_0$)/$p$)
end procedure REFTREE

procedure sum_weights($v_i$)
  summed_weight($v_i$) = weight($v_i$)
  if ($|C(v_i)| \neq 0$) then
    for $j$=1,$|C(v_i)|$
      summed_weight($v_i$) = summed_weight($v_i$) + sum_weights($c_j(v_i)$)
    end for
  endif
  return summed_weight($v_i$)
end procedure sum_weights

procedure make_partitions($v_i$, current_partition, current_sum, cutoff)
  new_sum = current_sum + summed_weight($v_i$)
  if (new_sum $\leq$ cutoff) then
    $v_i$ and all descendants are assigned to current_partition
    current_sum = new_sum
  else
    if ($|C(v_i)| = 0$) then
      current_partition = current_partition + 1
      cutoff = cutoff + summed_weight($v_0$)/$p$
      $v_i$ is assigned to current_partition
      current_sum = new_sum
    else
      for $j$=1,$|C(v_i)|$
        make_partition($c_j(v_i)$, current_partition, current_sum, cutoff)
      end for
    endif
  endif
end procedure make_partitions
```

Fig. 14. The REFTREE algorithm.

Authors personal copy

processors. However, the time is still well below 10% of the total running time of the adaptive-refinement multigrid-solution program, which was about 60 s.

Fig. 19 shows the time for computing each partition by each method for 32 processors. As expected, OCTREE and ParMETIS are the slowest methods. There is approximately a factor of 10 difference between the fastest method, HSFC, and the slowest. REFTREE is comparable to RIB and is in the middle of the pack, running 2–3 times slower than HSFC. Again the time for any of the methods is well below the total running time of the program.

Figs. 20 and 21 give an indication of the quality of the partitions as measured by the number of cut edges. Fig. 20 shows the average number of cut edges per partition, and Fig. 21 shows the maximum number of cut edges over all partitions. In general, the worst method is within a factor of 2 of the best method. For the average number of cut edges, REFTREE is generally the best method with ParMETIS coming in second. For maximum number of cut edges they switch roles. HSFC, which was the fastest method, generally produces the largest number of cut edges.

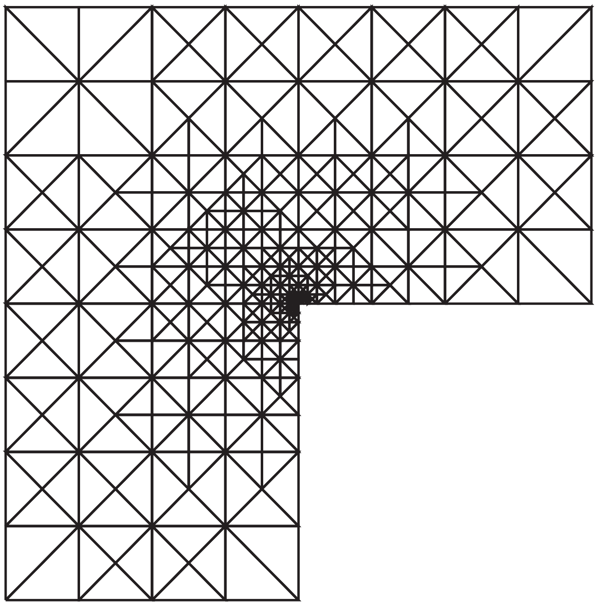

Fig. 15. The L shaped domain and an adaptive grid with 256 nodes.

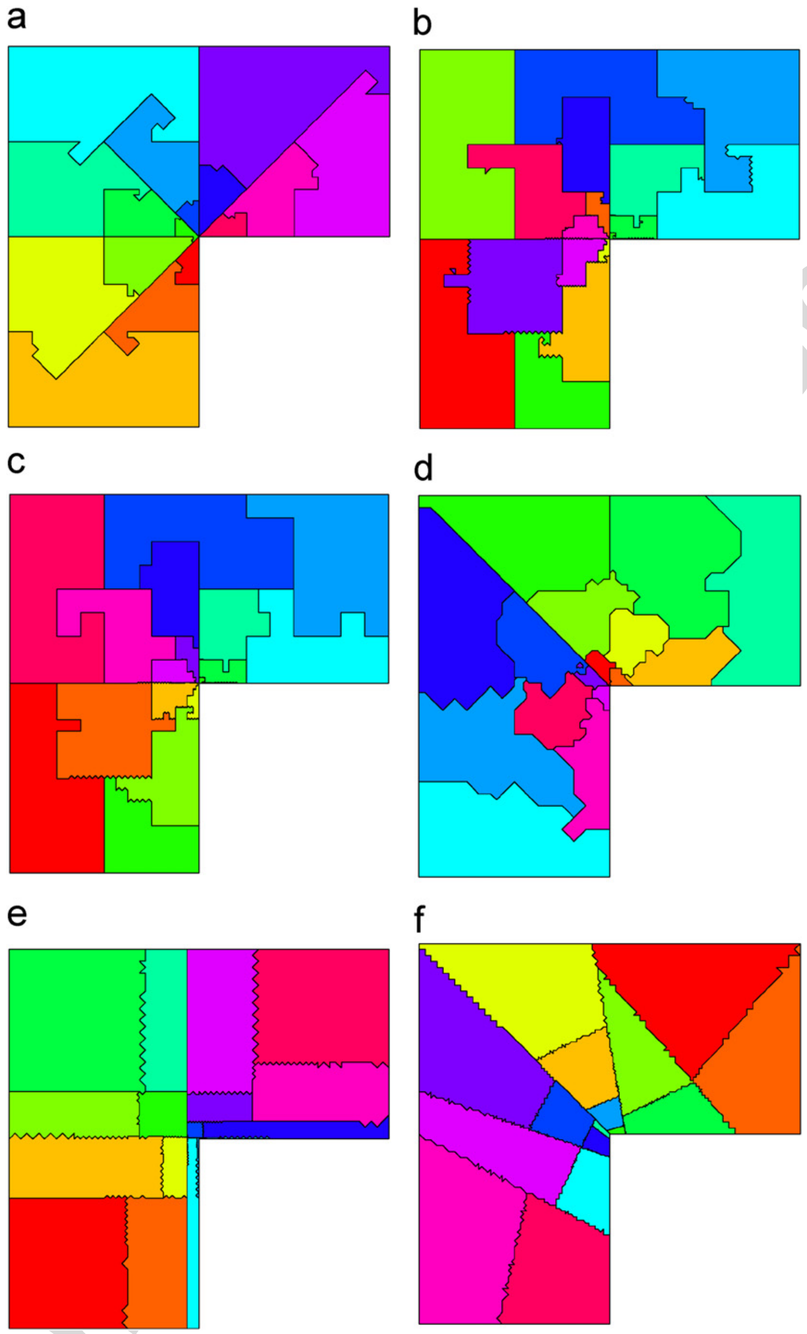

Fig. 16. Sample partitions for the L domain problem. (a) REFTREE, (b) HSFC, (c) OCTREE, (d) ParMETIS, (e) RCB, (f) RIB.

A second measure of partitioning quality is the number of elements that are moved from one partition to another. Table 1 gives the total number of elements that were transferred between partitions during the load balancing of the grids with about 250,000, 500,000 and 1,000,000 nodes. Here we see ParMETIS is clearly superior to the other methods. REFTREE is in the middle of the pack.

The second example solves Laplace's equation on a more complex domain. The domain and partitions for eight processors are shown in Fig. 22. This example illustrates that all of the other methods can produce disconnected partitions. In Fig. 22 REFTREE is the only method to produce connected partitions.

The same experiment was run for this problem as for the first example. The results are similar, and are summarized for 16 processors in Table 2. The amount of time used to compute partitions is given in column 2. Again HSFC is the fastest, ParMETIS and OCTREE the slowest, and the others are comparable. The quality of partitions as indicated by the maximum number of cut edges is best for ParMETIS with REFTREE second, and HSFC and OCTREE the worst. For the number of elements moved while load balancing the grid with 1,000,000

Author's personal copy

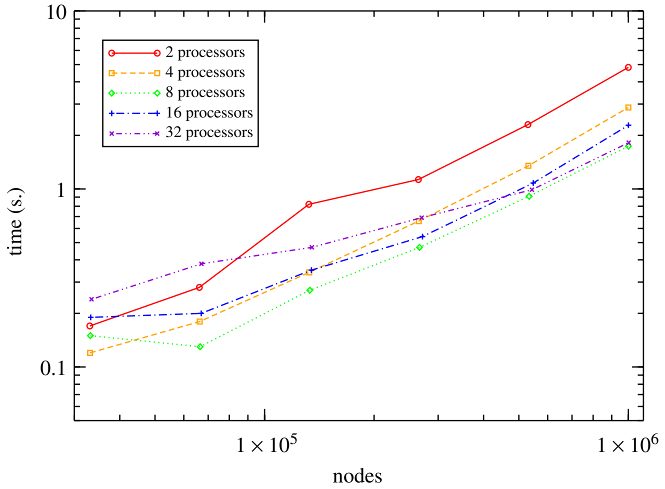

Fig. 17. Time for REFTREE partition with L domain example.

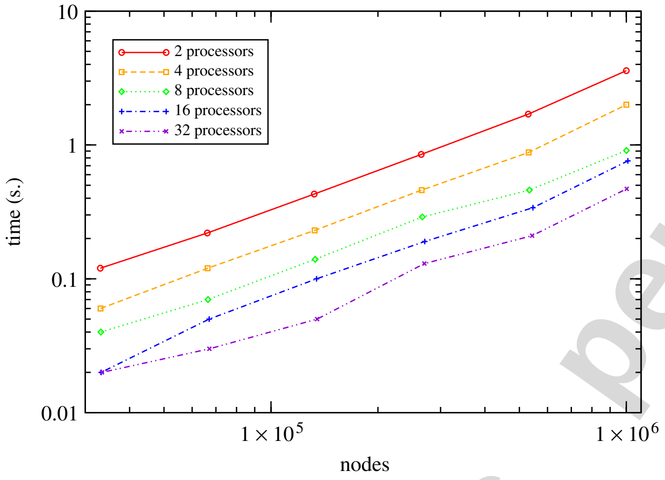

Fig. 18. Time for REFTREE partition with L domain example without time spent on communication.

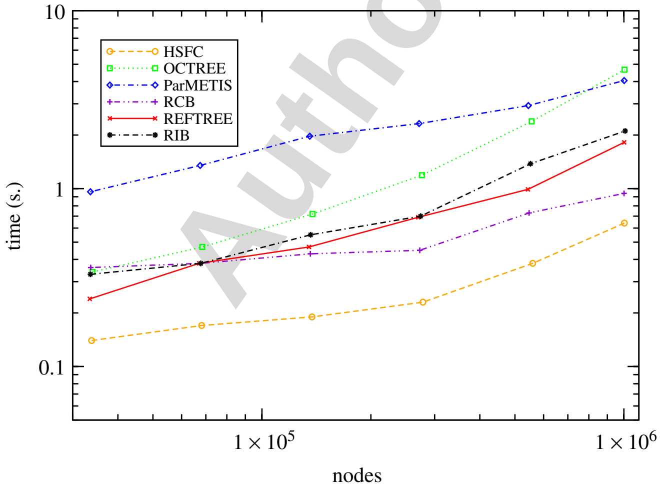

Fig. 19. Time for each method to partition L domain example into 32 partitions.

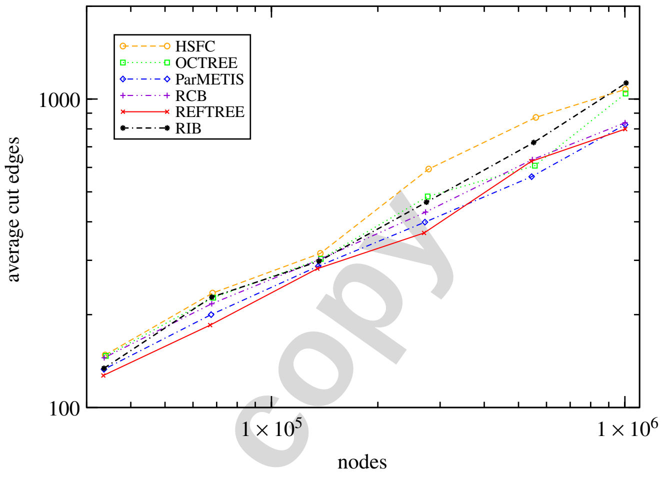

Fig. 20. Average number of cut edges for each method with 32 partitions of the L domain.

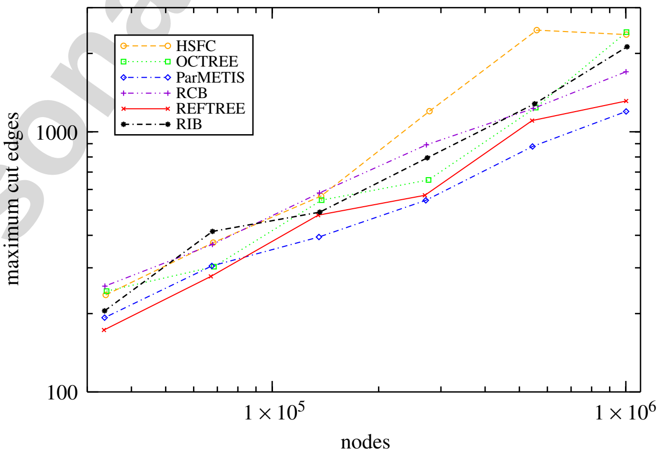

Fig. 21. Maximum number of cut edges for each method with 32 partitions of the L domain.

Table 1
Number of elements transferred

Method	250 K	500 K	1 M
REFTREE	58,829	186,078	213,588
HSFC	128,719	205,851	362,722
OCTREE	36,796	89,504	135,243
ParMETIS	7307	9962	32,097
RCB	50,563	61,600	96,463
RIB	125,665	291,553	669,757

nodes, ParMETIS is best and REFTREE is in the middle of the pack.

The third example examines the partitioning algorithms for a three-dimensional hexahedral grid with refinement in a half sphere of radius $\frac{1}{4}$ centered at the center of the top of a cube. This grid was not created in the context of solving a PDE, but was artificially created by refining elements that have a corner in the half sphere. A sample grid with 2164 elements is shown in Fig. 23, and partitions for eight processors with each of the

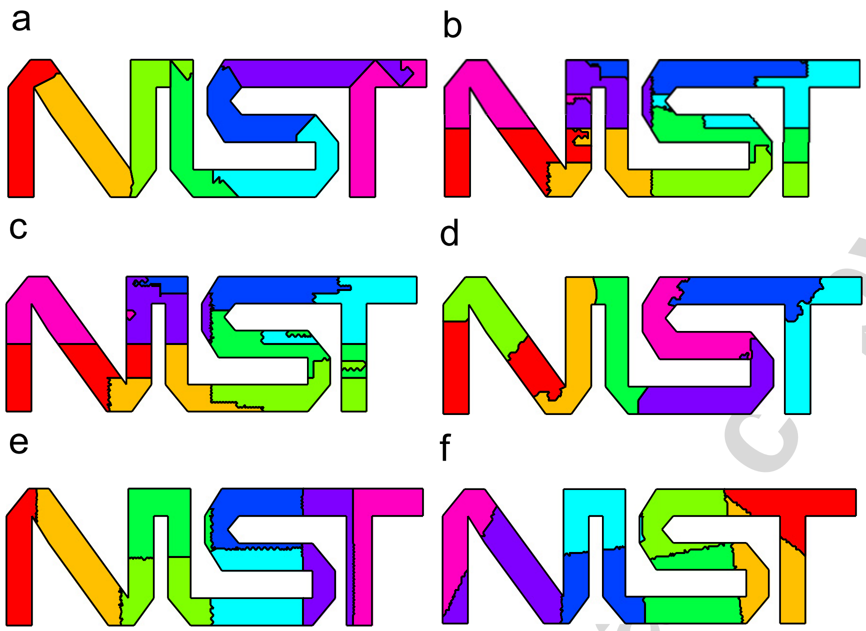

Fig. 22. Partition in eight sets for a complex domain. (a) REFTREE, (b) HSFC, (c) OCTREE, (d) ParMETIS, (e) RCB, (f) RIB.

Table 2
Results for 16 processors with the NIST domain

Method	Time (s)	Cuts	Moved
REFTREE	4.80	2607	298,206
HSFC	2.45	5529	360,824
OCTREE	8.69	7016	464,946
ParMETIS	7.40	1788	154,424
RCB	4.61	3072	269,166
RIB	5.56	2828	510,559

methods is shown in Fig. 24. In these figures, the cube has been cut down the middle and opened up to show the grid and partitions in the interior of the cube.

For this example, the program begins with a grid consisting of one cube and refines it once sequentially to a grid with 36 elements. It then alternately partitions and refines the grid. When the program terminates there are 4,605,840 elements. Runs were performed with 16 processors. The weights were 1.0 for leaf elements and 0.0 for non-leaf elements. The same measurements were made as in the first example.

Table 3 summarizes the results for the three-dimensional example. It reports the amount of time to partition the final grid, the maximum number of cut edges in the final partition, and the number of elements moved by the final partition. Again HSFC is the fastest and OCTREE is the slowest. REFTREE and ParMETIS are also much slower than the fastest methods. One of the reasons that REFTREE performs poorer on the three-dimensional problem than it did on the two-dimensional problems is that the determination of the child order is more complicated with octasected hexahedra and takes a significant portion of the time, while it is insignificant for bisected triangles. For number of cut edges, REFTREE is the best method, with RCB and OCTREE fairly close. ParMETIS again has the fewest elements moved, with RCB coming in second and REFTREE nearly identical to RCB. HSFC moved the most.

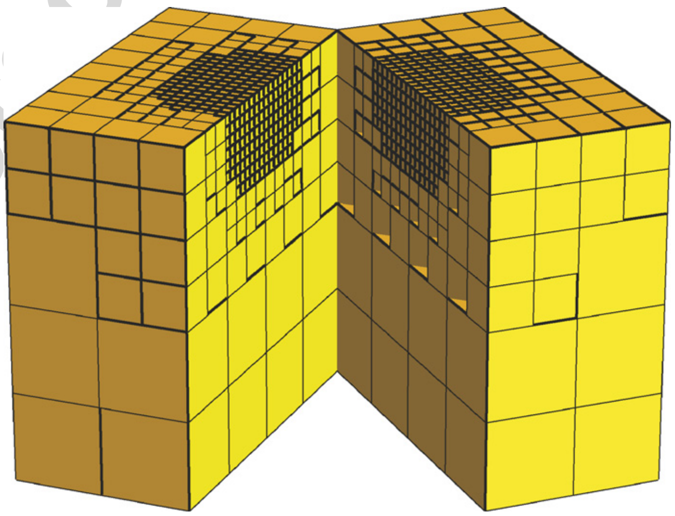

Fig. 23. An example grid for the three-dimensional domain.

6. Conclusion

This paper introduced the refinement-tree partitioning method (REFTREE) for grids that were created by adaptive refinement. It is closely related to the OCTREE method and SFC methods. REFTREE uses a tree representation of the refinement process with weights representing the amount of work associated with each element. The method applies to almost all types of elements and refinement strategies in two

Author's personal copy

Author's personal copy

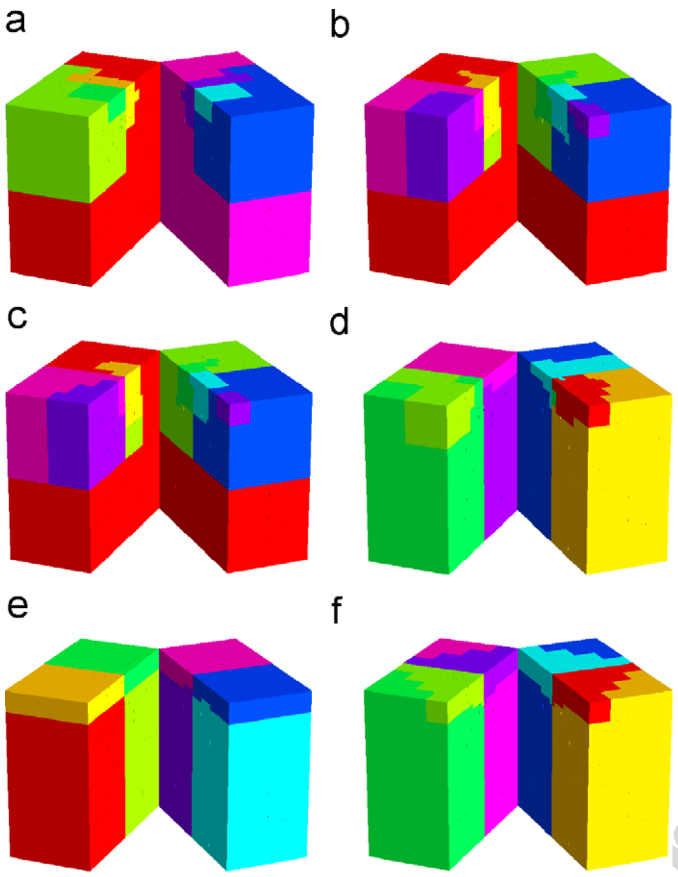

Fig. 24. Partition in eight sets for a three-dimensional domain. (a) REFTREE, (b) HSFC, (c) OCTREE, (d) ParMETIS, (e) RCB, (f) RIB.

Table 3
Results for 16 processors with the three-dimensional example

Method	Time (s)	Cuts	Moved
REFTREE	36.02	27,835	102,362
HSFC	3.82	48,019	1,772,104
OCTREE	45.10	31,171	151,455
ParMETIS	31.03	45,213	62,384
RCB	6.79	29,230	101,727
RIB	12.29	45,364	1,178,204

and three dimensions. For triangles and tetrahedra, it is guaranteed to produce connected partitions, which is not true of other partitioning methods. For all other applicable element types and refinement strategies it will produce connected partitions if a path through the elements of the initial grid can be found. When executed in parallel on p processors, the expected number of operations for partitioning into p sets is $O(N/p)$ with only one communication step. However, the communication step involves all-to-all communication which can become dominant for a large number of processors. Numerical results were presented in two and three dimensions. These results showed that in two dimensions REFTREE runs as quickly as methods like recursive coordinate bisection and produces high-quality partitions like multilevel diffusion. In three dimensions, REFTREE was slower, but still produced high-quality partitions.

References

[1] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, W.F. Mitchell, M. St. John, C. Vaughan, Zoltan: data-management services for parallel applications, ⟨http://www.cs.sandia.gov/Zoltan/⟩.

[2] K. Devine, B. Hendrickson, E. Boman, M. St. John, C. Vaughan, W.F. Mitchell, Zoltan: a dynamic load-balancing library for parallel applications, User's Guide, Sandia Technical Report SAND99-1377, Sandia National Laboratories, Albuquerque, NM, 2000.

[3] H.C. Edwards, A parallel infrastructure for scalable adaptive finite element methods and its application to least squares C^∞ collocation, Ph.D. Dissertation, University of Texas at Austin, 1997.

[4] J.E. Flaherty, R.M. Loy, M.S. Shephard, B.K. Szymanski, J.D. Teresco, L.H. Ziantz, Adaptive local refinement with Octree load-balancing for the parallel solution of three-dimensional conservation laws, J. Parallel Distributed Comput. 47 (1998) 139–152.

[5] G. Karypis, K. Schloegel, V. Kumar, ParMETIS: parallel graph partitioning and sparse matrix ordering library, Version 3.1, ⟨http://www-users.cs.umn.edu/~karypis/metis/parmetis/⟩.

[6] W.F. Mitchell, Adaptive refinement for arbitrary finite element spaces with hierarchical bases, J. Comput. Appl. Math. 36 (1991) 65–78.

[7] W.F. Mitchell, The refinement-tree partition for parallel solution of partial differential equations, NIST J. Res. 103 (1998) 405–414.

[8] W.F. Mitchell, The design of a parallel adaptive multi-level code in Fortran 90, in: Proceedings of the 2002 International Conference on Computational Science, 2002.

[9] W.F. Mitchell, PHAML, ⟨http://math.nist.gov/phaml⟩.

[10] W.F. Mitchell, Hamiltonian paths through two- and three-dimensional grids, NIST J. Res. 110 (2005) 127–136.

[11] A. Patra, J.T. Oden, Problem decomposition for adaptive *hp* finite element methods, Comput. Systems Engng. 6 (1995) 97.

[12] J.R. Pilkington, S.B. Baden, Dynamic partitioning of non-uniform structured workloads with spacefilling curves, IEEE Trans. Parallel Distributed Systems 7 (1996) 288–300.

[13] H. Sagan, Space-Filling Curves, Springer, New York, 1994.

[14] K. Schloegel, G. Karypis, V. Kumar, Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes, Technical Report #97-014, Department of Computer Science, University of Minnesota, 1997.

[15] M.S. Warren, J.K. Salmon, A parallel hashed Oct-Tree N-body algorithm, in: Proceedings of Supercomputing '93, Portland, OR, November 1993.

William F. Mitchell received his Ph.D. in computer science from the University of Illinois in 1988. He is currently a computer scientist in the Mathematical and Computational Sciences Division of the National Institute of Standards and Technology. His research interests are in numerical algorithms for the solution of partial differential equations, including adaptive grid refinement, multigrid methods, and parallel computing.