A Parallel Algorithm for Adaptive Local Refinement of Tetrahedral Meshes Using Bisection*

Lin-bo ZHANG
State Key Laboratory of Scientific and Engineering Computing,
Institute of Computational Mathematics and Scientific/Engineering Computing,
Academy of Mathematics and Systems Science,
Chinese Academy of Sciences,
Beijing 100080, P. R. China
zlb@lsec.cc.ac.cn

Preprint ICM–05–09, March 22, 2005

Abstract

Local mesh refinement is one of the key steps in implementations of adaptive finite element methods. This paper presents a parallel algorithm for distributed memory parallel computers for adaptive local refinement of tetrahedral meshes using bisection. The algorithm is part of PHG, *Parallel Hierarchical Grid*, a toolbox under development for parallel adaptive multigrid solution of PDEs. The algorithm proposed is characterized by allowing simultaneous refinement of submeshes to arbitrary levels before synchronization between submeshes and without the need of a central coordinator process for managing new vertices. Some general properties on local refinement of conforming tetrahedral meshes using bisection are also discussed which are useful in analysing and validating the parallel refinement algorithm as well as in simplifying the implementation.

Keywords: adaptive refinement, bisection, tetrahedral mesh, parallel algorithm, MPI
AMS subject classifications: 65Y05, 65N50

1 Introduction

The local refinement algorithms of simplicial meshes, mainly meshes composed of triangles in two dimensions, tetrahedra in three dimensions, have been extensively studied by many authors. In adaptive finite element computations, two types of refinement algorithms are widely used: the regular refinement and the bisectioning refinement. The regular refinement consists of simultaneously bisecting all edges of the triangle or tetrahedron to be refined, producing 4 smaller triangles or 8 smaller tetrahedra. Since the regular refinement cannot generate locally refined conforming meshes, either special numerical algorithms are designed to handle the hanging nodes, or it is combined with other types of refinement (the red/green algorithm) to produce a conforming mesh, see for examples [17, 18, 19]. While with the bisectioning refinement, only one edge of the triangle or the tetrahedron, called the *refinement edge*, is bisected, producing 2 smaller triangles or tetrahedra, as illustrated by Fig. 1. The main advantage of bisectioning refinement is that it can produce locally refined conforming meshes and nested finite element spaces. The main problems with bisectioning refinement are how to select the refinement edge such that triangles or tetrahedra produced by successive refinements do not degenerate, and the refinement procedure on a given mesh terminates in a finite number of steps, producing a conforming mesh which is ready for further refinements.

The methods for selecting the refinement edge proposed by various authors can be classified into two categories, namely the *longest edge approach* and the *newest vertex approach* (the latter is also called the *newest node approach* by some authors).

The longest edge based algorithms are proposed and studied by Rivara *et al* [12, 13]. In these algorithms, triangles or tetrahedra are always bisected using one of their longest edges, the finite termination of the refinement procedure is obvious because when traversing from one simplex to another in order to make the

*Supported by the 973 Program of China Grant No. 2005CB321702 and National Natural Science Foundation of China Project 10531080

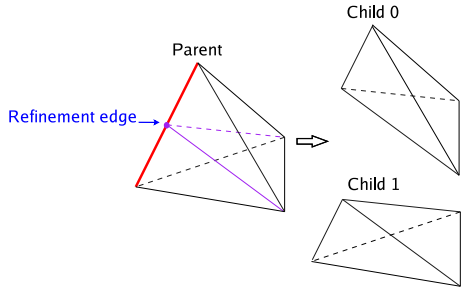

Figure 1: Bisection of a tetrahedron

resulting mesh conforming, one steps along paths of simplices with longer longest edges. In two dimensions, it can be shown that triangles produced by repeated bisection using the longest edge approach have angles bounded away from 0 and π. But in three dimensions, The non-degeneracy of tetrahedra produced by repeated bisections is still open.

The newest vertex approach was first proposed for two dimensional triangular meshes by Sewell [5], and was generalized to three dimensions by Bänsch [4]. Maubach [8] further generalized the method for $n+1$-simplicial meshes in n dimensions. More recent work on the newest vertex algorithms is described in the papers of Kossaczky [1], Liu and Joe [3], and Arnold *et al* [2]. Though the concept of the newest vertex approach is very simple in two dimensions: once the refinement edge for the triangles in the initial mesh is determined, the refinement edge of a newly generated triangle is the edge opposite to its newest vertex, its generalization to three dimensions is highly non-trivial, and the algorithms proposed by various authors are essentially equivalent, but use different interpretations. It is theoretically proved that tetrahedra generated by these algorithms belong to a finite number of similarity cases, which ensures non-degeneracy of tetrahedra produced by repeated bisections.

There are also works in the literature on parallel mesh refinement algorithms for triangular meshes using bisection. Rivara *et al* [14] proposed a parallel algorithm for global refinement of tetrahedral meshes which is not suitable for adaptive local refinement. Pébay and Thompson [15] presented a parallel refinement algorithm for tetrahedral meshes based on edge splitting. Jones and Plassmann [9] proposed and studied a parallel algorithm for adaptive local refinement of two dimensional triangular meshes. Barry, Jones, and Plassmann [10] also presented a framework for implementing parallel adaptive finite element applications and demonstrated it with 2D and 3D plasticity problems. Castaños and Savage [11] described the parallel algorithm for adaptive local refinement of tetrahedral meshes used in the PARED package.

The algorithm proposed here uses the newest vertex approach, because the newest vertex approach has the advantage of being purely topological, i.e., the refinement process only depends on the topological structure of meshes. The refinement edge of the descendant tetrahedra is uniquely determined once the coarsest mesh is properly initialized. Liu and Joe [3] pointed out that in the experiments they had carried out, much fewer tetrahedra in the final mesh were produced with the newest vertex algorithm than with a longest edge algorithm, and the resulting meshes of the latter depended heavily on roundoff errors due to comparison of edge lengths.

We end this introduction by briefly describing the algorithms of [2], [1] and [3] from which the present algorithm is derived.

The algorithms in [2, 3] are essentially the same. We use the terminologies of [2] to describe the algorithm because they are simpler. In this algorithm, each tetrahedron in the mesh has a marked edge which is its refinement edge. Each face also has a marked edge. For a face containing the refinement edge of a tetrahedron, the marked edge of the face is just the refinement edge. Tetrahedra in a marked mesh are classified into five types according to the relative position of their marked edges and a flag: type A (adjacent), type P_u (planar unflagged), type P_f (planar flagged), type O (opposite), and type M (mixed). For any marked tetrahedron, the marking of its children is precisely defined based on its type and marked edges through a set of well defined rules. Thus the refinement edges of all tetrahedra are well defined once the tetrahedra in the initial mesh are properly marked. Tetrahedra of types O and M can only exist in the initial mesh. A general algorithm for marking tetrahedra of an arbitrary initial mesh was also proposed in [2, 3] for which finite termination of the refinement procedure and the conformity of the resulting mesh are proved.

The algorithm of Kossaczky [1] is based on a different and interesting point of view: a tetrahedron to be bisected is embedded into a reference parallelepiped consisting of 6 congruent tetrahedra sharing one

diagonal of it, successive bisections of the tetrahedra are then defined by bisecting in turn the diagonal of the parallelepiped shared by all tetrahedra, the diagonals of faces of the parallelepiped shared by some tetrahedra, and the edges of the parallelepiped. After three levels of bisections, the parallelepiped is divided into 8 smaller parallelepipeds, all similar to each other, then the same process can be restarted with the smaller parallelepipeds. Using this point of view it is easy to see the finiteness of similarity cases of tetrahedra produced by repeated bisections, and the refinement procedure is well defined once the embeddings for the tetrahedra in the initial mesh are defined. Based on this idea, Kossaczky developed an algorithm in which each tetrahedron is associated with a type, and the refinement edge of a tetrahedron is denoted by the local numbering of its four vertices, with edge 0–1 being the refinement edge. The type and numbering of vertices of the new tetrahedra are derived from their parent. There are three types of tetrahedra according to the position of their refinement edge in the reference parallelepiped: type *Diagonal* (the refinement edge is the diagonal of the reference parallelepiped), type *Face Diagonal* (the refinement edge is a diagonal of a face of the reference parallelepiped), and type *Edge* (the refinement edge is an edge of the reference parallelepiped). It is easy to verify that this algorithm is equivalent to the algorithm of [2] by mapping type Diagonal to type A, type Face Diagonal to type P_u, type Edge to type P_f, and define the marked edge of the faces according to local numbering of vertices.

In our implementation, Kossaczky's notation is used for storing refinement information of a tetrahedron, with which only the type of the tetrahedron needs to be explicitly stored. To define types and ordering of vertices for the tetrahedra in an arbitrary initial mesh (called *initial embedding* by Kossaczky), we first use the initial marking algorithm described in [2] to mark all tetrahedra, then map tetrahedra of types A and P_u to types Diagonal and Face Diagonal respectively, and renumber the vertices of the tetrahedra accordingly. In order to handle tetrahedra of types O and M which are not defined in Kossaczky's algorithm, two extra types, that we still call type *Opposite* and type *Mixed*, respectively, are introduced. Fig. 2 summarizes the 5 types of tetrahedra and associated bisection rules used in our implementation, in which refinement edges are indicated with thick lines. The last three rules in Fig. 2 were given in [1] and the first two rules can be easily derived by examining refinement rules in [2].

2 The Serial Refinement Algorithm

In this section, we summarize some basic properties previously known of the bisectioning refinement based on the newest vertex approach, and prove a property which shows independence of the resulting mesh on the actual implementation. For simplicity of discussions, we first introduce some formal notations and definitions.

A tetrahedron t is defined by its four vertices, and t is *non degenerate* if the four vertices are not co-planar. For convenience of discussions, the variable t will be used to denote both a tetrahedron (as an object) and its interior domain, whichever is applicable.

Let t be a tetrahedron, then $\mathcal{E}(t)$ denotes the set of the 6 edges of t, $\mathcal{F}(t)$ the set of the 4 faces of t, and $\mathcal{V}(t)$ the set of the 4 vertices of t.

Definition 1. *Let Ω be an open bounded domain in $\mathbb{R}^3$. A tetrahedral mesh T on Ω is a set of non degenerate tetrahedra satisfying the following conditions:*

1. $\overline{\Omega} = \bigcup_{t \in T} \overline{t}$.
2. $\forall t, t' \in T$, *either* $t = t'$ *or* $t \cap t' = \emptyset$.

As with a tetrahedron, $\mathcal{E}(T)$, $\mathcal{F}(T)$, and $\mathcal{V}(T)$ are used to denote the set of edges, faces, and vertices of a tetrahedral mesh, respectively. Let $f \in \mathcal{F}(T)$, f is called a *boundary face* if $f \subset \partial\Omega$, an *interior face* otherwise.

Let t be a tetrahedron, a bisection of t consists of dividing t into two smaller tetrahedra by inserting a new vertex at the center of a selected edge, called the *refinement edge*, and linking the new vertex to the two vertices of the edge opposite to the refinement edge, as shown by Fig. 1. The two new tetrahedra are called *children* of t. All tetrahedra generated by repeated bisections of a tetrahedron t and t itself are called *descendants* of t. The *generation* of descendants of a tetrahedron t is defined as follows: t has generation 0 and if t' is a generation k descendant, then the two children of t' have generation $k+1$. We will denote by $t' \leq t$ if t' is a descendant of t (and $t' < t$ if $t' \neq t$). A tetrahedron t is called a descendant of a tetrahedral mesh T if it is a descendant of a tetrahedron of T.

Let T and T' be two tetrahedral meshes on a domain Ω, if all tetrahedra of T' are descendants of T, then T' is called a *descendant mesh* of T. As for the case of tetrahedra, the notation $T' \leq T$ (or $T' < T$ if $T' \neq T$) is used to denote that T' is a descendant mesh of T.

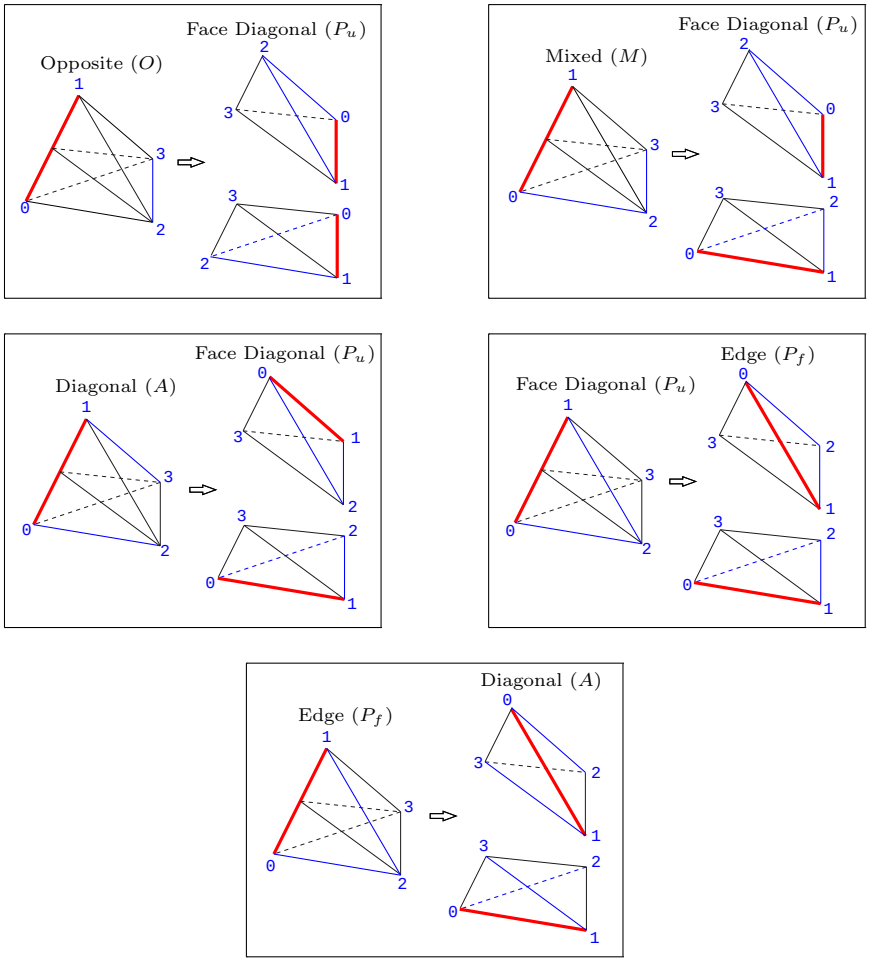

Figure 2: Refinement rules for the 5 refinement types

For convenience of descriptions, we introduce the concept of a *bisection operator*. For a given tetrahedral mesh T, a bisection operator on T can be thought of the procedure of bisecting selected tetrahedra of T and the attributes attached to T which define an unique, pre-determined refinement edge for each tetrahedron of T as well as for all its descendants. (Note that with this concept, the method for marking an initial mesh is considered an attribute of the bisection operator.)

A bisection operator is denoted by R and the following conventions will be used:

- $R(t)$ denotes the submesh consisting of the two children of t.
- $R^k(t)$ denotes the submesh consisting of all descendants of t of generation k.
- $R(\mathbf{t}) \triangleq R^k(t)$, $R(T, \mathbf{t}) \triangleq R(T, t, k) \triangleq (T \setminus \{t\}) \bigcup R(\mathbf{t})$, where $\mathbf{t} = (t, k) \in T \times \mathbb{N}$ and $\mathbb{N}$ denotes the set of positive integers.
- $R(T, t) \triangleq (T \setminus \{t\}) \bigcup R(t)$.
- $R(T, S) \triangleq (T \setminus S) \bigcup\limits_{t \in S} R(t)$, where S is a subset of T.
- $R(T, \mathbb{S}) \triangleq (T \setminus S) \bigcup\limits_{\mathbf{t} \in \mathbb{S}} R(\mathbf{t})$, where $\mathbb{S} \subset T \times \mathbb{N}$ and $S = \{t \mid \exists k, (t, k) \in \mathbb{S}\}$. $\mathbb{S}$ is called a *selection* of T since it selects the set of tetrahedra to be refined and defines their refinement levels.

It is clear that for any selections $\mathbb{S} \subset T \times \mathbb{N}$, $R(T, \mathbb{S})$ is a descendant mesh of T.

Definition 2. *A tetrahedral mesh T is* conforming *if $\forall t \in T$ and $\forall f \in \mathcal{F}(t)$, either f is a boundary face, or $\exists t' \in T$, $t' \neq t$ and $f \in \mathcal{F}(t')$.*

Note: the definition of conformity here is narrower than its usual meaning. More precisely it refers to *face conformity*. It does not ensure conformity of edges. An example of a mesh consisting of 3 tetrahedra which is conforming by Definition 2 but has a non conforming edge is shown in Fig. 3. Such meshes are not common in the context of finite element or finite volume approximations. For performance reasons, the parallel refinement algorithm presented here works with face neighbours, which only ensures face conformity. A similar algorithm can be designed to work with edge neighbours for ensuring both face and edge conformity, but it requires much more memory for storing neighbour information and need more communications in the parallel implementation. Hereafter throughout this paper, except when explicitly stated, the term *conformity* will always refer to face conformity.

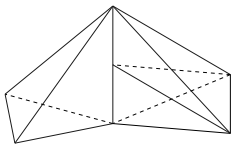

Figure 3: A face-conforming mesh with a non conforming edge

Definition 3. *Let T be a tetrahedral mesh, R a bisection operator. R is called a* conforming bisection operator *if, for any descendants t and t' of T sharing a common face f, let e and e' be the refinement edge of t and t' respectively, if f contains both e and e', then $e = e'$.*

The conformity of a bisection operator corresponds to the concept *conformingly marked tetrahedral mesh* in [2]. It is a necessary condition for the existence of conforming descendant meshes for an arbitrary selection $\mathbb{S}$.

In the subsequent discussions, when not explicitly stated, we will assume that a conforming tetrahedral mesh is always associated with an underlying conforming bisection operator.

Definition 4. *Let t be a tetrahedron of a tetrahedral mesh T and f a face of T. f is called a* hanging face *of t in T if $\exists f' \in \mathcal{F}(t)$, $f \neq f'$ and $f \subset f'$.*

Definition 5. *Let T be a conforming tetrahedral mesh, $\mathbb{S} \subset T \times \mathbb{N}$ a selection of T, R a bisection operator on T, T' a conforming descendant mesh of T. T' is called a* conforming refinement of T with respect to $\mathbb{S}$ *if $T' \leq R(T, \mathbb{S})$.*

Now we describe the serial refinement algorithm. It is slightly different from the algorithm `LocalRefine` in [2] in order to cover the parallel refinement procedure defined later in Section 4.

Algorithm 1. *The input is a conforming tetrahedral mesh T and a selection $\mathbb{S}$. The output is a conforming refinement T' of T with respect to $\mathbb{S}$.*

Step 0 *Let $k = 0$, $T_0 = R(T, \mathbb{S})$*

Step 1 *Let* $S_k = \{t \mid t \in T_k$ *and* t *has a hanging face in* $T_k\}$

Step 2 *If* $S_k = \emptyset$ *then let* $T' = T_k$ *and terminate.*

Step 3 *Let* $T_{k+1} = R(T_k, S'_k)$*, where* S'_k *is an arbitrary non empty subset of* S_k*.*

Step 4 *Let* $k = k + 1$ *and goto step 1.*

In our implementation, the bisection operator is defined by the types and local ordering of vertices of tetrahedra in the mesh, which are determined by Arnold's initial marking algorithm and the refinement rules shown in Fig. 2. We will call this bisection operator the *newest vertex bisection operator*. It is well defined and is conforming, and for any given selection Algorithm 1 terminates in a finite number of steps and generates a conforming descendant mesh, see [2, 3, 1] for discussions and proofs. Though the original proofs in [2, 3, 1] were based on edge conformity, it is not difficult to verify that they are also valid with face conformity.

Definition 6. *Let* T *be a conforming tetrahedral mesh,* $\mathbb{S}$ *a selection of* T*, The* canonical refinement *of* T *with respect to* $\mathbb{S}$*, denoted by* $T_C(\mathbb{S})$*, is defined to be the descendant mesh of* T *satisfying the following conditions:*

1. $T_C(\mathbb{S})$ *is a conforming refinement of* T *with respect to* $\mathbb{S}$*.*
2. *If* T' *is a conforming refinement of* T *with respect to* $\mathbb{S}$*, then* $T' \leq T_C(\mathbb{S})$*.*

The following theorem gives existence and uniqueness of the canonical refinement.

Theorem 1. *Let* T *be a conforming tetrahedral mesh,* $\mathbb{S}$ *a selection of* T*. If a conforming refinement of* T *with respect to* $\mathbb{S}$ *exists, then* $T_C(\mathbb{S})$ *exists and is unique.*

Proof. The uniqueness of the canonical refinement is obvious from its definition and the fact that for any descendant meshes T' and T'' of T, if both $T' \leq T''$ and $T'' \leq T'$ hold, then $T' = T''$.

Denote by $\mathscr{R}(T, \mathbb{S})$ the set of all conforming refinements of T with respect to $\mathbb{S}$. Let $\mathscr{T}(T, \mathbb{S})$ be the set of all tetrahedra of $\mathscr{R}(T, \mathbb{S})$. Define $T_C(\mathbb{S})$ to be the subset of $\mathscr{T}(T, \mathbb{S})$ consisting of tetrahedra of smallest generations, i.e.:

$$T_C(\mathbb{S}) = \{t \mid t \in \mathscr{T}(T, \mathbb{S}) \text{ and } t \text{ does not have an ancestor in } \mathscr{T}(T, \mathbb{S})\}$$

It is not difficult to verify that $T_C(\mathbb{S})$ is well defined since $\mathscr{R}(T, \mathbb{S}) \neq \emptyset$, and is a descendant mesh of T (i.e., $T_C(\mathbb{S})$ satisfies the conditions in Definition 1). Thus to prove $T_C(\mathbb{S})$ is the canonical refinement of T with respect to $\mathbb{S}$, we just need to prove it's conforming.

Suppose $t \in T_C(\mathbb{S})$, $f \in \mathcal{F}(t)$ and f is not a boundary face, we need to show that t has a neighbour in $T_C(\mathbb{S})$ sharing the face f.

By definition of $T_C(\mathbb{S})$ there exists $T' \in \mathscr{R}(T, \mathbb{S})$ such that $t \in T'$. Since T' is conforming, $\exists t' \in T'$, $t' \neq t$, such that $f \in \mathcal{F}(t')$. If $t' \in T_C(\mathbb{S})$ then t' is the neighbour we are looking for. Otherwise, by definition of $T_C(\mathbb{S})$, there exist T'' and t'', $T'' \in \mathscr{R}(T, \mathbb{S})$, $t'' \in T'' \bigcap T_C(\mathbb{S})$, such that $t' < t''$. The proof is concluded if we can show that $f \in \mathcal{F}(t'')$. Let's suppose the contrary, then f is a descendant face of a face, say f'', of t'' and $f \neq f''$, let t''' be the neighbour of t'' in T'' sharing the face f'', then it is easily seen that $t < t'''$, which contradicts the definition of $T_C(\mathbb{S})$. □

Theorem 2. *Let* T *be a conforming tetrahedral mesh associated with the newest vertex bisection operator,* $\mathbb{S}$ *a selection of* T*, then the resulting mesh of Algorithm 1 is the canonical refinement* $T_C(\mathbb{S})$*.*

Proof. First, since Algorithm 1 terminates in a finite number of steps for a given choice of S'_k, $k = 0, 1, \ldots$, and produces a conforming refinement of T with respect to $\mathbb{S}$, by Theorem 1 the canonical refinement $T_C(\mathbb{S})$ exists.

Next, we prove by induction that for any given choice of S'_k, $k = 0, 1, \ldots$, $T_C(\mathbb{S}) \leq T_k$. Obviously, $T_C(\mathbb{S}) \leq T_0$ holds. Suppose for some k, $T_C(\mathbb{S}) \leq T_k$ holds and $S_k \neq \emptyset$, then for any $t \in T_C(\mathbb{S})$, $\exists t' \in T_k$ such that t is a descendant of t'. We distinguish the following two cases:

case 1: If $t' \notin S'_k$, then $t' \in T_{k+1}$.

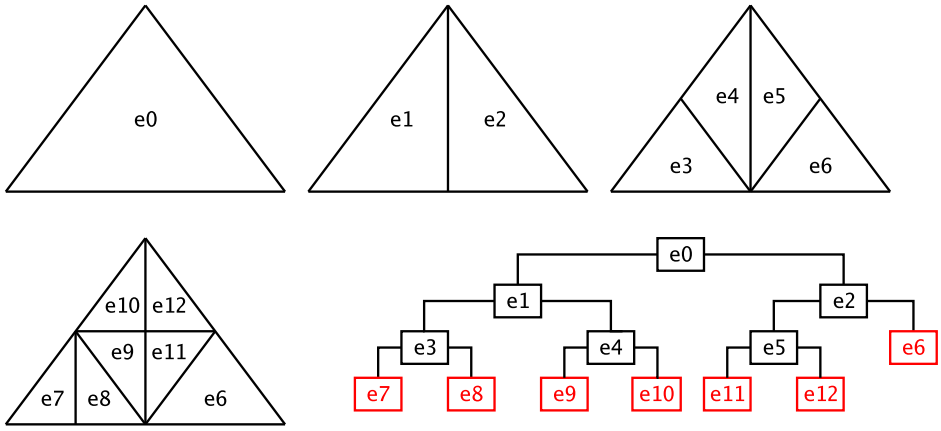

Figure 4: Example of a hierarchical binary tree

case 2: If $t' \in S'_k$, then t' has a hanging face in T_k. In order to show t is a descendant of T_{k+1}, we only need to show $t < t'$. Let's suppose the contrary, i.e., $t = t'$, then t has a hanging face in T_k, since by the induction hypothesis $T_C(\mathbb{S})$ is a descendant mesh of T_k, it is not difficult to see that t also has a hanging face in $T_C(\mathbb{S})$, which contradicts the conformity of $T_C(\mathbb{S})$.

in both cases, t is a descendant of T_{k+1}, thus we have $T_C(\mathbb{S}) \leq T_{k+1}$. □

Corollary 1. *The resulting mesh produced by Algorithm 1 is independent of the choice of the subsets S'_k in step 2.*

Theorem 2 and Corollary 1 ensure that for any selection, the resulting mesh of Algorithm 1 only depends on the initial marking, not on the processing order, which is an inherent property of the bisectioning refinement algorithm, as long as a consistent, pre-determined rule is used for selecting the refinement edge which does not depend on the processing order.

In the implementation, to support mesh coarsening (unrefinement) and multigrid algorithms, all tetrahedra created by successive bisections are stored as a binary tree, in which the leaf nodes constitute the finest mesh. Mesh coarsening is necessary in some situations, for example, when solving time dependent problems, and its parallel implementation is a non trivial and interesting subject of study, but here we will focus on the mesh refinement algorithm. Fig. 4 is an illustration of a binary tree for an adaptively refined 2D triangular mesh, in which the initial mesh contains one triangle, and the finest mesh contains 7 triangles.

3 Mesh Partitioning

The parallel algorithm is based on the standard message passing interface MPI. In order to distribute a mesh T on a distributed memory parallel computer, it is partitioned into P submeshes T_i, $i = 0, \ldots, P-1$, where P is the number of MPI processes. The partitioning is computed using existing tools like METIS [16]. In our implementation the partitioning is element-based, i.e., the set of tetrahedra of T is divided into P disjoint subsets. After mesh partitioning, in each process is stored a binary tree, which we call a subtree, whose leaf elements constitute a submesh. Each subtree contains all ancestors of the leaf elements and is an incomplete binary tree in the sense that some non leaf nodes may only have one branch, with the other branch stored in another subtree. Each leaf element only exists in one subtree, but non leaf elements may be shared across multiple subtrees. Fig. 5 shows the subtrees after partitioning the mesh in Fig. 4 into 2 submeshes. In our implementation no ghost elements are stored, because in finite element applications most computations which require neighbourhood traversal can be replaced with element-based mesh traversal. For example, when computing the matrix of a linear system, the mesh is traversed element by element, contribution from each element to the matrix is computed independently and added to the global matrix through an unified linear solver interface. In special situations where neighbourhood traversal is really needed, ghost elements can be dynamically created.

In a distributed mesh, there are three kinds of indices for a vertex: its index within a tetrahedron, in the range 0 – 3; its index within a submesh, in the range $0 - n_v - 1$, where n_v is the number of vertices in the

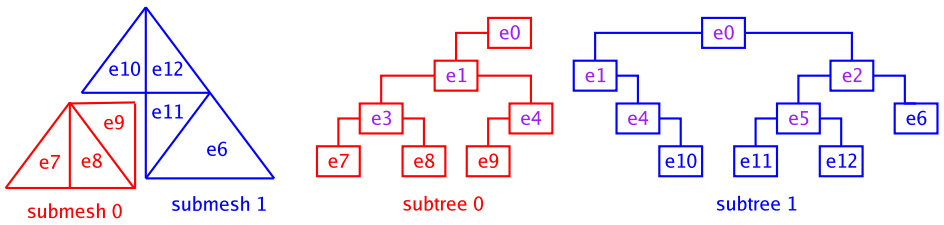

Figure 5: Subtrees for a partitioned mesh

submesh; and its index within the whole mesh, in the range $0 - N_v - 1$, where N_v is the number of vertices in the whole mesh. To simplify the discussions below, we will call these 3 kinds of indices *t-index*, *l-index*, and *g-index*, respectively.

In the implementation, in each tetrahedron are stored an array `v[`i`]`, $i = 0, 1, 2, 3$, which are the l-indices of the 4 vertices of the tetrahedron, and the type of the tetrahedron. The array `v` and the type of a tetrahedron uniquely determine its refinement edge, and bisecting a tetrahedron simply consists of computing the array `v` and the type of its children according to the set of rules in Fig. 2.

For each submesh, an array called `L2Gmap` is used to store the g-indices of all vertices of the submesh. More precisely, `L2Gmap[`i`]` gives the g-index of the vertex with l-index i. If the mesh is not partitioned, then `L2Gmap[`i`]` $= i$, i.e., the l-indices and g-indices are identical (in this case `L2Gmap` need not be explicitly stored). The array `L2Gmap` is created when the mesh is partitioned, and updated during mesh refinement, coarsening, and repartitioning. The algorithm for updating `L2Gmap` after the mesh is refined will be discussed later in Section 5.

A neighbour of a tetrahedron t may be on another submesh, such a neighbour is called a *remote neighbour* of t, and their common face is called a *shared face*. Similarly, a vertex may be either exclusively owned by a submesh or shared by two or more submeshes, it is called a *private vertex* in the former case and a *shared vertex* in the latter case.

A remote neighbour is stored in a C struct named `RNEIGHBOUR` which contains essentially five members: `local`, `remote`, `rank`, `vertex`, and `rface`. `local` is the pointer (or index) to the local tetrahedron, `remote` is the pointer (or index) to the remote tetrahedron in the remote submesh, `rank` is the MPI rank of the process containing the remote submesh, `vertex` is the t-index in the local tetrahedron of the vertex opposite to the shared face, and `rface` is a three-entry array which stores the t-indices of the vertices of the shared face in the remote tetrahedron. Entries of the array `rface` are ordered in the following way: let $v_0 < v_1 < v_2$ be the t-indices in the local tetrahedron of the three vertices of the shared face, then `rface[0]`, `rface[1]` and `rface[2]` are their corresponding t-indices in the remote tetrahedron. The array `rface` plays a key role in the implementation of our parallel refinement algorithm.

4 The Parallel Refinement Algorithm

The basic idea of the parallel refinement algorithm is a natural one, which was also used by other parallel adaptive local refinement algorithms based on bisection [9, 11]. In our implementation, for a given selection $\mathbb{S}$, the refinement procedure is divided into two phases. In the first phase, the submeshes are refined independently, with the shared faces treated as if they were boundary faces. This step creates locally conforming submeshes, but with non conforming shared faces. In the second phase, tetrahedra containing one or more shared faces which have been bisected during the first phase are exchanged between neighbour submeshes, and tetrahedra having one or more hanging shared faces are bisected, the process is repeated until the global conformity of the mesh is reached, i.e., no more hanging shared faces exist.

Algorithm 2. *Parallel adaptive local refinement algorithm using bisection.*

Step 1 *perform Algorithm 1 in each submesh with the given selection.*

Step 2.1 *terminate if no non conforming shared faces.*

Step 2.2 *exchange information between submeshes about shared faces which have been bisected in Step 1 or Step 2.3.*

Step 2.3 *bisect in all submeshes tetrahedra having non conforming shared faces using Algorithm 1 until all non conforming shared faces are bisected and local conformity is obtained.*

Step 2.4 *goto Step 2.1.*

In Step 1 of Algorithm 2, tetrahedra selected for refinement are bisected to requested levels, and local conformity of submeshes is obtained. The steps 2.1–2.4 are called the *synchronization loop* which ensures global conformity of the resulting mesh.

In the unrefined mesh, i.e., the mesh before performing Step 1 of Algorithm 2, each submesh has a list of `RNEIGHBOUR` structs with correctly linked data between submeshes, we call this list the *primary list of remote neighbours*, or shortly the *primary list*. During the synchronization loop two more lists are created. The first one is used to save the `RNEIGHBOUR` structs for new tetrahedra, as well as information necessary to track from a tetrahedron in the unrefined mesh to any of its descendants (described below), it is called the *temporary list of remote neighbours*, or shortly the *temporary list*. The second one is used to save `RNEIGHBOUR` structs with updated information for new tetrahedra, it is called the *saved list of remote neighbours*, or shortly the *saved list*.

In Algorithm 2, each time a tetrahedron having shared faces is bisected, new `RNEIGHBOUR` structs are created for its children having shared faces, one for each shared face. The `local` member of the new `RNEIGHBOUR` struct points to corresponding new tetrahedron and the other members are inherited from the parent (using data from the primary list). Thus at the time when a new tetrahedron is created, only the `local` and the `rank` members of its `RNEIGHBOUR` struct contain correct information, the other members are simply copied from its ancestor in the unrefined mesh and will be updated later. The new `RNEIGHBOUR` structs are added both to the primary and temporary lists. For each new `RNEIGHBOUR` struct in the temporary list, three extra members are saved:

1. A `depth` member which is the relative generation of the new tetrahedron with respect to its ancestor in the unrefined mesh.

2. A `path` member which contains bitwise flags whose first `depth` bits indicate which child to choose for each bisection when going from the ancestor in the unrefined mesh to the new tetrahedron, 0 means child 0, 1 means child 1. In the implementation a 32-bit or 64-bit integer is used to store this member, it allows a tetrahedron in the unrefined mesh to be repeatedly bisected 32 or 64 times in Algorithm 2, which is more than enough for practical applications.

3. A `type` member which is the type of the ancestor in the unrefined mesh of the new tetrahedron.

Before giving details of the implementation, we give a proposition which is needed for justifying the steps described below. It can be easily verified by examining bisection of tetrahedra of all possible types (it was stated as Lemma 1 in [1]).

Proposition 1. *Let t be a tetrahedron. Let f be any face of t. If f does not contain the refinement edge of t, then f must contain the refinement edge of one of the children of t.*

At the beginning of a synchronization loop, each process sends entries in the temporary list to corresponding processes and flushes the temporary list for further new entries. Then the entries received from other processes are processed. Suppose $\{\texttt{rn}, \texttt{depth}, \texttt{path}, \texttt{type}\}$ is an entry received where `rn` is the `RNEIGHBOUR` struct, it is processed as follows (the function `bisect(`t`)` below means applying Algorithm 1 to the submesh with the selection $\{t\} \times \{1\}$):

1. Let t be the tetrahedron pointed by `rn.remote`, which is in the unrefined mesh, and let `rn0` be the corresponding `RNEIGHBOUR` struct.

2. Let t_r be the matching remote tetrahedron (pointed by `rn.local` in the remote submesh).

3. If the shared face of t_r is not bisected then goto 10.

4. If the shared face contains the refinement edge of t then goto 8

5. If t has not been bisected, then `bisect(`t`)`.

6. Let t' be the child of t containing the shared face, set $t := t'$. Now the shared face of t contains the refinement edge of t according to Proposition 1.

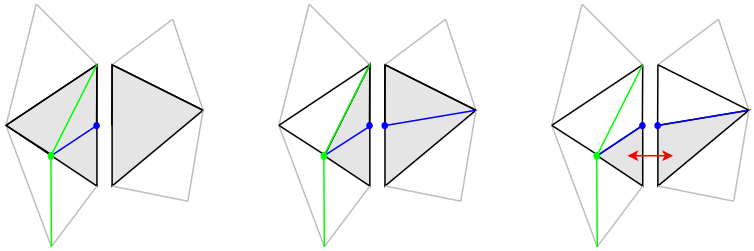

Figure 6: Updating a matching pair of elements

7. Update the members of `rn` and `rn0` accordingly.
8. If t has not been bisected, then `bisect(`t`)`.
9. Among the children of t, choose the one matching the refinement path of the remote tetrahedron (the decision on which child to choose can be derived from current values of `rface` and `vertex` members of `rn` and `rn0`, `path&1`, and the types of the local and remote tetrahedra), and set t to that child.
10. Similarly set t_r to the child belonging to the current refinement path in the remote submesh.
11. Update the members of `rn` and `rn0` to match the new t and t_r.
12. Set `depth` := `depth` -1, `path` := `path>>1`, and set `type` to the type of t_r.
13. If `depth` > 0 then goto 1
14. If t has been refined in the current submesh but the shared face has not, then set t to the child containing the shared face, and update the members of `rn` and `rn0` accordingly.
15. Append `rn0` to the saved list.

The main idea in the above steps is traversing along the refinement path in both submeshes from the two matching tetrahedra in the unrefined mesh to the two matching tetrahedra in the refined mesh, bisecting tetrahedra as necessary, and updating information stored in corresponding `RNEIGHBOUR` structs. It is illustrated in Fig. 6 with a two dimensional mesh.

After all received entries are processed by the above steps, they are sent back to where they came, and appended to the saved list. Then the synchronization loop is restarted with new entries in the temporary list created during the above steps.

Note that in the synchronization loop entries in the primary list are kept unchanged, up to date data are saved in the saved list. This is important because otherwise one has to deal with `RNEIGHBOUR` structs in some intermediate states which lead to very complex situations.

The synchronization loop terminates if all the temporary lists for all submeshes are empty after the above steps. Then entries in the saved list and the primary list are used to construct the list of `RNEIGHBOUR` structs for the final new mesh. If multiple entries exist for a same shared face, the one with a larger `depth` value is used.

It can be verified that Algorithm 2 is equivalent to an implementation of Algorithm 1 for the global mesh. Thus the theoretical results of Section 2 apply to Algorithm 2, ensuring the finite termination of the synchronization loop and global conformity of the resulting mesh. It can also be verified that after the above steps the new `RNEIGHBOUR` structs contain correct information about the remote neighbours in the final mesh.

5 Assigning Global Indices to New Vertices

Assignment of global indices consists of updating the `L2Gmap` array, which is postponed to after the synchronization loop for increasing communication granularity. In this section we describe our algorithm for updating the `L2Gmap` array.

During the refinement process, when a new vertex is created, an unique l-index is assigned to it. The l-indices assigned to new vertices are strictly increasing with respect to their order of creation. If a new

```
function comp_vertex(i, j)
if i < N_v or j < N_v then
  return i − j
else
  Find the triplet (a_0, b_0, m_0) in the list of shared vertices such that m_0 = i − N_v,
    if the entry is not found then set m_0 := −1.
  Find the triplet (a_1, b_1, m_1) in the list of shared vertices such that m_1 = j − N_v,
    if the entry is not found then set m_1 := −1.
  if m_0 = −1 or m_1 = −1 then
    if m_0 ≠ −1 then
      return 1
    end if
    if m_1 ≠ −1 then
      return −1
    end if
    return i − j
  else
    Let r := comp_vertex(a_0, a_1)
    if r ≠ 0 then
      return r
    else
      return comp_vertex(b_0, b_1)
    end if
  end if
end if
end
```

Figure 7: The function `comp_vertex`

vertex is a shared vertex, then it is saved to a list. Each entry in the list is stored as a triplet $\{a, b, m\}$, where m is the l-index of the new vertex, a and b correspond to the two vertices of the refinement edge. Since only shared vertices need to be saved, the size of the list is relatively small compared to total number of new vertices. Since a tetrahedron may be bisected arbitrary number of times during the refinement process, a or b may be either old or new vertices (here *old vertex* means a vertex in the unrefined mesh).

To define values of a and b, we introduce the notation of *c-index* (which means a compound index). Let v be the l-index of a vertex, then its c-index c is defined as:

$$c = \begin{cases} \texttt{L2Gmap[}v\texttt{]} & \text{if } v \text{ is an old vertex} \\ v + N_v & \text{if } v \text{ is a new vertex} \end{cases}$$

where N_v is the number of vertices in the unrefined global mesh. For an old vertex, the c-index is just its g-index, and for a new vertex, the c-index is its l-index plus the number of vertices in the unrefined global mesh. Let x and y be the c-indices of the two vertices, if `comp_vertex`$(x, y) < 0$, then $a := x$ and $b := y$, otherwise $a := y$ and $b := x$. The recursive function `comp_vertex` which compares two c-indices is defined in Fig. 7. By examining the procedure for assigning g-indices to new vertices described later in this section, one can verify that the sign of the value returned by `comp_vertex` is exactly the same as comparing the g-indices of the two vertices. This function is very useful when one needs to compare or order vertices according to their g-indices during the refinement procedure before `L2Gmap` is updated.

The following proposition, Proposition 2, validates the function `comp_vertex`, as well as the function `comp_shared_vertex` defined later in Figure 8.

Proposition 2. *Let $\{a, b, m\}$ be an entry in the list of new shared vertices. If a is a new vertex, i.e., $a \geq N_v$, then there exists an entry $\{a_0, b_0, m_0\}$ in the list such that $m_0 = a - N_v$ and $m_0 < m$. Similarly, If b is a new vertex ($b \geq N_v$), then there exists an entry $\{a_1, b_1, m_1\}$ in the list such that $m_1 = b - N_v$ and $m_1 < m$. This property holds at any time during the refinement procedure.*

Proof. Let $\{a, b, m\}$ be a triplet in the list of new shared vertices, it is clear that both a and b must be shared vertices. Moreover, if a or b is a new vertex, since it is created before the vertex m, a corresponding entry must exist in the list. □

```
function comp_shared_vertex(i, j)
Let (a_0, b_0, m_0) be the triplet for the entry i.
Let (a_1, b_1, m_1) be the triplet for the entry j.
Let r := comp_shared_vertex0(a_0, i, a_1, j)
if r ≠ 0 then
  return r
else
  return comp_shared_vertex0(b_0, i, b_1, j)
end if
end

function comp_shared_vertex0(a, i, b, j)
if a < N_v or b < N_v then
  return a − b
else
  Let B_{k_i} be the block containing the i-th entry of the list,
    find the triplet (a_0, b_0, m_0) in B_{k_i} such that m_0 = a − N_v.
  Let B_{k_j} be the block containing the j-th entry of the list,
    find the triplet (a_1, b_1, m_1) in B_{k_j} such that m_1 = b − N_v.
  Let r := comp_shared_vertex0(a_0, i, a_1, j)
  if r ≠ 0 then
    return r
  else
    return comp_shared_vertex0(b_0, i, b_1, j)
  end if
end if
end
```

Figure 8: The function `comp_shared_vertex`

After termination of the synchronization loop, each process sends its list of new shared vertices, together with the number of new private vertices, to all other processes (it consists of an `MPI_Allgatherv` operation). Then each process has an identical copy of the lists of new shared vertices and the numbers of private vertices of all submeshes, and can update its `L2Gmap` independently.

Assigning g-indices to private vertices is quite easy and straightforward. Let the number of new private vertices in submesh i be l_i, $i = 0, \ldots, P-1$, then $L = \sum_{i=0}^{P-1} l_i$ is the total number of new private vertices. The g-indices of the new private vertices are in the range $N_v, \ldots, N_v + L - 1$ ($N_v, \ldots, N_v + l_0 - 1$ for submesh 0, $N_v + l_0, \ldots, N_v + l_0 + l_1 - 1$ for submesh 1, etc.).

Assigning g-indices to shared vertices is more complicated. Let $m_0, \ldots, m_{P-1}$ be the numbers of new shared vertices from all submeshes, $M = \sum_{i=0}^{P-1} m_i$ the total number of new shared vertices. Denote by B_k ($k = 0, \ldots, P-1$) the list of new shared vertices from submesh k. The lists B_k are concatenated into a list of size M (which is in fact the result of the `MPI_Allgatherv` operation). A corresponding list of integers, `index`, whose entries are indices to the list of new shared vertices, is created and initialized with `index[`i`]` $= i$, $i = 0, \ldots, M-1$. The `index` list is then sorted using the function `comp_shared_vertex` defined in Fig. 8. The function `comp_shared_vertex` takes as input two indices to the list of shared vertices and returns an integer value reflecting the result of comparison of the two new shared vertices.

After the `index` list is sorted, new g-indices are continuously assigned to new shared vertices according to the entries of `index`, starting from $N_v + L$. Two entries for which the return value of `comp_shared_vertex` is 0 represent a same vertex and are assigned a same g-index value.

6 Numerical Experiments

We present three numerical examples demonstrating the parallel refinement algorithm.

In the first example, the domain is the unit cube and the initial mesh contains 6 congruent tetrahedra. The mesh is repeatedly refined, each time about 25% tetrahedra are randomly selected for refinement. At the beginning, the serial algorithm is run on process 0 a number of times until the total number of tetrahedra

Table 1: Results of the first example

Lenovo DeepComp 1800, 2GHz Pentium IV/Myrinet 2000				
Nprocs	# tetrahedra	Wall time (s)	Tetrahedra/sec	Efficiency
1	1,311,940	2.22	591,747	1.00
2	2,580,736	2.52	1,025,958	0.87
3	4,624,250	3.50	1,322,531	0.74
4	4,624,856	2.64	1,753,354	0.74
6	9,898,356	3.97	2,490,235	0.70
8	9,893,362	3.09	3,205,731	0.68
12	19,764,128	4.33	4,567,630	0.64
16	19,774,100	3.32	5,959,549	0.63
24	35,773,120	4.65	7,691,865	0.54
32	35,778,384	3.60	9,931,217	0.52
48	76,666,264	5.48	13,999,528	0.49
64	76,752,286	4.49	17,079,386	0.45
SGI Origin 3800, 600MHz MIPS R14000				
Nprocs	# tetrahedra	Wall time (s)	Tetrahedra/sec	Efficiency
1	1,313,888	4.21	312,260	1.00
2	2,584,392	4.67	553,793	0.89
3	4,624,648	6.91	669,250	0.71
4	4,623,142	5.11	905,508	0.72
6	9,897,970	9.02	1,097,188	0.59
8	9,889,678	5.68	1,740,701	0.70
12	19,762,892	9.45	2,090,727	0.56
16	19,768,314	6.48	3,052,394	0.61
24	35,737,554	10.23	3,494,356	0.47
32	35,717,938	7.97	4,480,648	0.45
48	76,603,682	10.62	7,215,973	0.49
60	76,540,224	9.30	8,232,786	0.45

in the mesh exceeds 10,000, at that point the mesh is partitioned into P submeshes using the METIS function `METIS_PartMeshDual` and distributed onto the P processes. Then the parallel refinement algorithm is started with the P processes and is repeated until the number of tetrahedra in each submesh exceeds one million. Since in this example the numbers of tetrahedra in different submeshes are roughly the same, no mesh repartitioning is needed. The number of tetrahedra in the final mesh, the parallel refinement time (not including the serial refinement part), and the average number of tetrahedra created per second with various number of processes are given in Table 1, The speedup and parallel efficiency obtained with this example are shown in Fig. 9. By observing Fig. 9 we see that the performance of the refinement algorithm scales as P^{α}. Using the data in Table 1 one gets by least square fit $\alpha \approx 0.82$ for the DeepComp 1800, and $\alpha \approx 0.80$ for the Origin 3800. Thus, for example, the parallel efficiency is expected to be between 25% and 29% on 1024 processes for these kinds of parallel computers, which is quite satisfactory for us.

In the second example, the domain is $(0,1)^3 \setminus (\frac{1}{2},1)^3$. i.e., the unit cube with one of its corners removed. The initial mesh contains 42 tetrahedra. In each refinement, tetrahedra which touch the spherical surface $(x-\frac{1}{2})^2+(y-\frac{1}{2})^2+(z-\frac{1}{2})^2=(\frac{3}{5})^2$ are selected for refinement, thus the tetrahedra in the final mesh concentrate towards the surface. Fig. 10 shows the initial mesh, a refined mesh with 42,546 tetrahedra, and a partitioned mesh. The serial refinement algorithm is first repeatedly performed to bring the number of tetrahedra to 16,044, then the mesh is partitioned and distributed, and the parallel refinement algorithm is repeatedly performed on the distributed mesh, until the final mesh, which contains 13,044,234 tetrahedra, is obtained. In this example, if no mesh repartitioning or redistribution is performed, the load imbalance factor LIF (the maximum number of tetrahedra over the average number of tetrahedra in the submeshes) in the final mesh varies between 1.1 and 2.0 for different number of processes. Two sets of tests have been run. In the first set of tests, after each mesh refinement, if the LIF exceeded 1.1, then a new mesh partitioning was computed using the ParMETIS function `ParMETIS_V3_AdaptiveRepart` and the submeshes were redistributed. In the second set of tests, no mesh redistribution was performed. Table 2 and Fig. 11 show the refinement time and parallel efficiency for various number of processes. For this example, since the size of the final mesh is

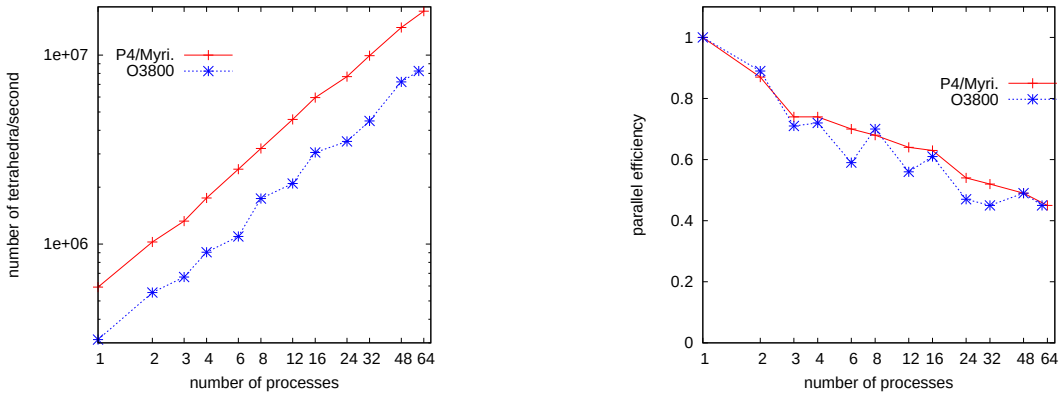

Figure 9: The first example

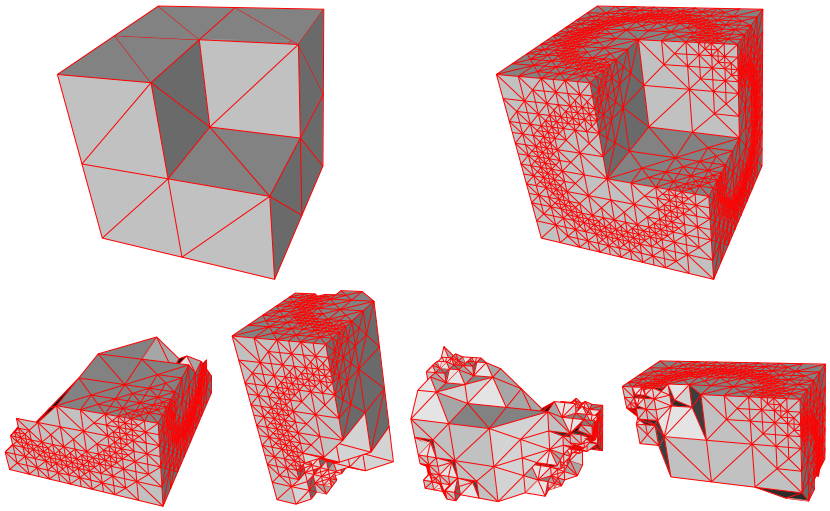

Figure 10: The second example

fixed, the sizes of the submeshes become smaller with increasing number of submeshes, which makes more efficient use of memory cache of the computers, resulting in super-linear speedups in some cases.

The mesh redistribution times in the second set of tests are given in Table 3, they include times spent in `ParMETIS_V3_AdaptiveRepart` and times for migrating and reconnecting branches of the subtrees. The time for the refinement step just before mesh redistribution is also given in the table. The redistribution times are typically one order of magnitude longer than correspinding refinement times. Our current mesh redistribution code is mainly a functional implementation which works reliably but has rather poor performance, which is a subject of study in our future work. More results on the mesh redistribution time will be given in the next example to show their influence in the context of adaptive finite element computations.

The third example consists of solving the Poisson equation $-\Delta u = f$ with Dirichlet boundary condition in the unit cube $(0,1)^3$. This is an example used in the adaptive finite element toolbox ALBERTA [21]. It was used to verify the suitability of our underlying distributed mesh infrastructure for adaptive finite element applications, and evaluate the performance of various parts of our code.

The equation is solved using Lagrange elements of orders 1 to 4. The initial mesh contains 5 tetrahedra. Elements are selected for refinement using the following a posteriori error estimator:

$$\eta_t^2 = h_t^2 \|\Delta u_h + f_h\|_{0;t}^2 + \frac{1}{2} \sum_{f \in \mathcal{F}(t), f \not\subset \partial\Omega} h_f \|[\bigtriangledown u_h \cdot \mathbf{n}_f]_f\|_{0;f}^2$$

where u_h denotes the finite element solution, h_t the diameter of the element t, h_f the diameter of the face f, $\mathbf{n}_f$ the unit normal vector of the face f, and $[\cdot]_f$ the jump across the face f. The exact solution $u(x,y,z) = e^{-10(x^2+y^2+z^2)}$ was used in the computations. The linear systems were solved using PETSc with

Table 2: Results of the second example

Lenovo DeepComp 1800, 2GHz Pentium IV/Myrinet 2000					
	With mesh redistribution		Without mesh redistribution		
Nprocs	Wall time (s)	Efficiency	Wall time (s)	Efficiency	LIF
4	8.30	1.00	8.30	1.00	1.03
6	5.19	1.07	5.19	1.07	1.04
8	3.92	1.06	4.24	0.98	1.12
12	2.57	1.08	2.62	1.06	1.12
16	1.98	1.05	2.10	0.99	1.20
24	1.40	0.99	1.42	0.97	1.15
32	1.10	0.94	1.28	0.81	1.37
48	0.93	0.74	1.04	0.67	1.50
64	0.87	0.60	1.01	0.51	1.78
SGI Origin 3800, 600MHz MIPS R14000					
	With mesh redistribution		Without mesh redistribution		
Nprocs	Wall time (s)	Efficiency	Wall time (s)	Efficiency	LIF
2	31.29	1.00	31.29	1.00	1.02
4	15.75	0.99	15.16	1.03	1.03
6	11.51	0.91	12.62	0.83	1.03
8	8.55	0.91	10.05	0.78	1.06
12	6.85	0.76	5.13	1.02	1.06
16	4.05	0.97	4.37	0.90	1.17
24	3.38	0.77	3.36	0.78	1.14
32	2.78	0.70	2.70	0.72	1.18
60	1.73	0.60	2.71	0.36	1.79

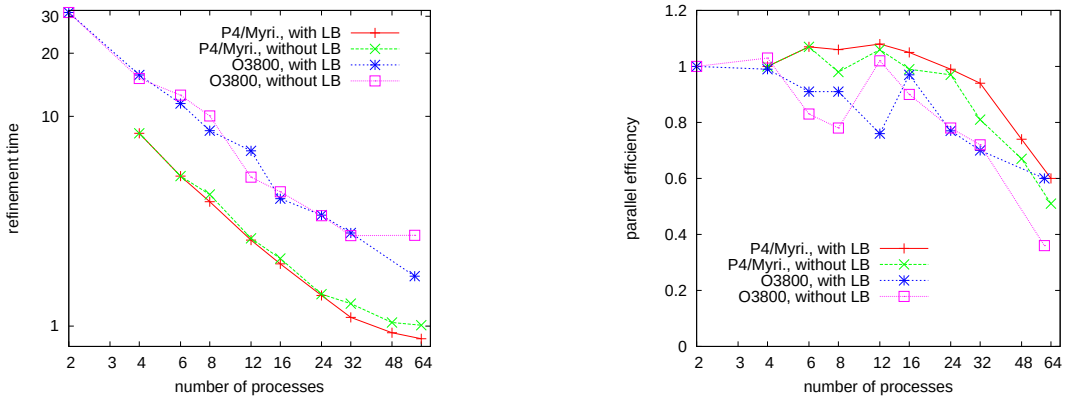

Figure 11: The second example

Table 3: Mesh redistribution time on Lenovo DeepComp 1800 for the second example, "refinement time" refers to the time spent in the refinement step prior to mesh redistribution, "ratio" is the ratio of mesh redistribution time to refinement time.

Nprocs	Nbr tetrahedra	Refinement time	Redistr. time	Ratio
8	789264	0.2080	1.4756	7.09
12	789264	0.1467	1.2353	8.42
16	411432	0.0481	0.5182	10.76
24	3226062	0.3177	3.2919	10.36
32	789264	0.0783	0.7646	9.76
48	3226062	0.2092	2.4773	11.84
64	13044234	0.6549	8.8240	13.47

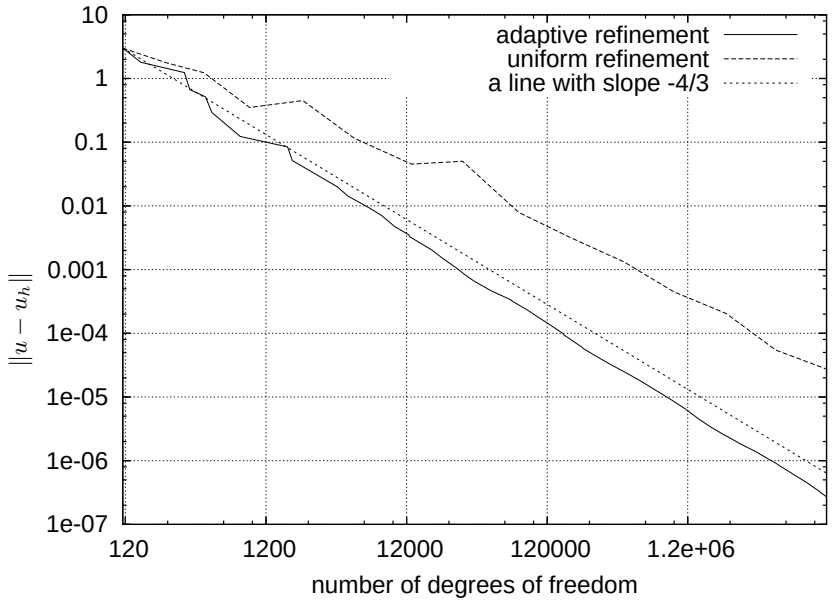

Figure 12: Convergence history of the 4-th order Lagrange element

its default solver (GMRES(30)) and preconditioner (ILU(0)) [22], and the `rtol` value was set to `1e-10`. In the adaptation loops, the numerical solution was interpolated to the new mesh after mesh refinement which was used as the initial solution of the linear system.

The computations were performed on the Lenovo DeepComp 1800. In each run, the compuation was started with one process, new processes were added when the everage number of elements per process exceeded 200 until all processes available were used, and submeshes were redistributed whenever the load imbalance factor exceeded 1.2.

Fig. 12 shows the convergence of the numerical solution for the 4-th order Lagrange element, for both uniform refinement and adaptive refinement.

Table 4 gives some timing results from the computations. In order to exhibit costs of mesh redistributions, only the times for the last adaptation step (one "solve⇒estimate⇒refine⇒redistribute" loop) in which mesh redistribution occurred were shown. The refinement times in the table include the time for interpolating u_h and evaluating f at new elements, and the redistribution times include the time for migrating finite element and geometric data (including barycentric Jacobian, normal vectors, diameters, etc.). Since the linear solver used was not the best one available for this problem, the solution times should only be regarded as a rough indication. The mesh redistribution time represents an important part in overall computational time, especially with low order elements, so it is important to reduce the number of mesh redistributions. One way to reduce the number of mesh redistributions is to predict refinement of elements using the a posteriori error estimates, and use this information when computing the mesh partitioning (see, for example, the work by Bank and Holst [20]).

7 Conclusion

A parallel algorithm for adaptive local refinement of tetrahedral meshes using bisection is presented. The algorithm uses the the newest vertex approach and is characterized by allowing simultaneous refinement of submeshes to arbitrary levels before synchronization between submeshes and without the need of a central coordinator process for managing new vertices. Independence of the resulting mesh on the processing order and mesh partitioning is proved using the concept of canonical refinement. The scalability of the refinement algorithm is satisfactory. Based on this algorithm, we have implemented the core part of our adaptive finite element toolbox PHG which consists of modules for mesh import and initialization, refinement, coarsening, and redistribution. We have also finished basic data structures and a linear solver interface for parallel adaptive finite element computations. Work is undergone to complete the functionality of the toolbox, such as support for *h-p* adaptivity and multigrid, and apply the toolbox to solve application problems, like the Maxwell's equations.

Table 4: Wall times (in seconds) in some adaptation steps of various parts of the computations. The numbers in parentheses following the solution times are the numbers of iterations.

Element	Nprocs	Number elements	Linear system		Estimate	Refinement	Redistribution
			Assembly	Solution			
P_1	16	2434024	1.67	3.44 (166)	1.48	1.41	9.34
	32	6721346	2.61	8.79 (269)	2.04	2.27	13.95
	64	35506914	6.41	41.63 (413)	5.37	6.19	13.95
	200	35506908	2.13	11.49 (423)	1.89	4.29	58.80
P_2	16	2786314	17.31	16.70 (47)	3.99	1.82	8.37
	32	5652030	17.90	17.60 (47)	4.03	2.47	12.06
	64	9752578	15.62	13.19 (40)	3.55	2.33	22.10
	200	16669782	9.00	6.36 (27)	2.14	2.08	49.59
P_3	16	173594	8.30	4.40 (37)	1.31	0.19	0.64
	32	2106842	53.97	15.02 (9)	6.39	1.07	4.20
	64	3174854	40.49	10.74 (6)	4.96	0.76	5.65
	200	3945754	15.93	4.94 (5)	3.08	0.58	6.70
P_4	16	63544	13.67	3.64 (13)	1.17	0.11	0.31
	32	425378	46.98	11.13 (5)	4.78	0.29	1.20
	64	571640	32.49	7.23 (4)	2.83	0.39	1.87
	200	485785	8.56	2.71 (4)	0.91	0.21	2.51

References

[1] I. Kossaczky, A Recursive Approach to Local Mesh Refinement in Two and Three dimensions, *J. Comput. Appl. Math.,* ***55*** *(1994), 275–288*

[2] D. N. Arnold, A. Mukherjee, and L. Pouly, Locally Adapted Tetrahedral Meshes Using Bisection, *SIAM J. Sci. Comput.,* ***22*** *(2000), No. 2, 431–448*

[3] A. Liu, B. Joe, Quality Local Refinement of Tetrahedral Meshes Based on Bisection, *SIAM J. Sci. Comput.,* ***16*** *(1995), No. 6, 1269–1291*

[4] E. Bänsch, An adaptive finite–element strategy for the three dimensional time dependent Navier-Stokes equations, *J. Comput. Appl. Math.,* ***36*** *(1991), 3–28.*

[5] E. G. Sewell, Automatic generation of triangulation for piecewise polynomial approximation, *Ph. D. Thesis, Purdue Univ., West Lafayette, IN, 1972*

[6] W. F. Mitchell, Unified multilevel adaptive finite element methods for elliptic problems, *Ph. D. Thesis, Report no. UIUCDCS–R–88–1436, Dept. Comput. Sci., Univ. Illinois, Urbana, IL, 1988*

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

[8] J. M. Maubach, Local bisection refinement for N–simplicial grids generated by reflection, *SIAM J. Sci. Comput.,* ***16*** *(1995), 210–227*

[9] M. T. Jones and P. E. Plassmann, Parallel Algorithms for Adaptive Mesh Refinement, *SIAM J. Sci. Comput.,* ***18*** *(1997), No. 3, 686–708*

[10] W. J. Barry, M. T. Jones, P. E. Plassmann, Parallel adaptive mesh refinement techniques for plasticity problems, Advances in Engineering Software, Volume 29, Number 3, April 1998, pp. 217-225(9)

[11] J. G. Castaños, J. E. Savage, Parallel Refinement of Unstructured Meshes, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, November 3–6, 1999, MIT, Boston, USA*.

[12] M. C. Rivara, Mesh refinement processes based on the generalized bisection of simplices, *SIAM J. Numer. Anal.,* ***21*** *(1984), 604–613*

[13] A. Plaza, M. C. Rivara, Mesh refinement based on the 8-tetrahedra longest-edge partition, *Proceedings, 12th International Meshing Roundtable, Sandia National Laboratories, 67–78, Sept. 2003*

[14] M. C. Rivara, D. Pizarro, N. Chrisochoides, Parallel refinement of tetrahedral meshes using terminal-edge bisection algorithm, *Proceedings 13th International Meshing Roundtable, Williamsbourg USA September 19–22 2004*

[15] Philippe P. Pébay and David C. Thompson, Parallel Mesh Refinement Without Communication *Proceedings 13th International Meshing Roundtable, Williamsbourg USA September 19–22 2004*

[16] METIS — Family of Multilevel Partitioning Algorithms,
`http://www-users.cs.umn.edu/~karypis/metis/`

[17] R. E. Bank, A. H. Sherman and A. Weiser, Refinement algorithms and data structs for regular local mesh refinement, *Scientific Computing, R. Stepleman et al., ed., IMACS/North-Holland Publishing Company, Amsterdam, 1983, pp. 3–17*

[18] Bey J., Tetrahedral Grid Refinement, *Computing, vol. 55, 355–378, 1995*

[19] de Cougny H. L., Shephard M. S., Parallel Refinement and Coarsening of Tetrahedral Meshes, *Int. J. for Num. Meth. in Eng., vol. 46, 1101–1125, 1999*

[20] R. E. Bank and M. Holst, A New Paradigm for Parallel Adaptive Meshing Algorithms, *SIAM J. Sci. Comput.,* ***22*** *(2000), pp. 1411–1443*

[21] Schmidt, Alfred, Siebert, Kunibert G., Design of adaptive finite element software: The finite element toolbox ALBERTA, *Springer LNCSE Series 42 (2005)*

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