PARALLEL MULTILEVEL TETRAHEDRAL GRID REFINEMENT

SVEN GROSS AND ARNOLD REUSKEN*

Abstract. In this paper we analyze a parallel version of a multilevel red/green local refinement algorithm for tetrahedral meshes. The refinement method is similar to the approaches used in the UG-package [33] and by BEY [11, 12]. We introduce a new data distribution format that is very suitable for the parallel multilevel refinement algorithm. This format is called an admissible hierarchical decomposition. We will prove that the application of the parallel refinement algorithm to an input admissible hierarchical decomposition yields an admissible hierarchical decomposition. The analysis shows that the data partitioning between the processors is such that we have a favourable data locality (e.g., parent and children are on the same processor) and at the same time only a small amount of copies.

AMS subject classifications. 65N50, 65N55

Key words. tetrahedral grid refinement, parallelization, stable refinement, consistent triangulations

1. Introduction.

In the field of numerical solution methods for partial differential equations adaptive discretization methods gain growing acceptance. In such adaptive methods both simplicial and hexahedral meshes are used. In this paper we only consider the former class and we restrict ourselves to the three dimensional situation, i.e., we treat tetrahedral meshes (which we also call triangulations). For well-known reasons one usually requires these meshes to fulfill a stability and consistency condition. The stability condition means that in a repeated refinement process, the interior angles of all tetrahedra that are generated must be uniformly bounded away from zero. A triangulation is called consistent if the intersection of any two tetrahedra from the triangulation is either empty, a common face, a common edge or a common vertex. Several refinement algorithms are known that satisfy both conditions for consistent input triangulations. These algorithms can be divided in two classes: *red/green refinement methods* (for example, [11, 13, 27, 35]) and *bisection methods* (for example, [1, 26, 28, 30, 31, 32]). For a discussion of the main differences between these classes of methods we refer to the literature (e.g., [11, 13, 32]). In this paper we only consider a *multilevel tetrahedral* red/green refinement method. The idea of a *multilevel* refinement (and coarsening) strategy was introduced by BASTIAN [6] and further developed in [8, 11, 13, 24, 25, 35]. This grid refinement technique is used in UG [33]; for an overview we refer to [7, 9]. Similar techniques are used in several other packages, for example, in KASKADE [23], DROPS [18], PML/MG [29].

To be able to summarize some interesting properties of a multilevel refinement algorithm we first have to introduce a few notions. A sequence of triangulations $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ of some domain Ω is called a *multilevel triangulation* if: 1. each tetrahedron in $\mathcal{T}_k$ ($k \geq 1$) is either in $\mathcal{T}_{k-1}$ or it is obtained by a (red or green) refinement of its parent in $\mathcal{T}_{k-1}$; 2. if a tetrahedron in $\mathcal{T}_k$ is not refined when going to $\mathcal{T}_{k+1}$, then it remains unrefined in all $\mathcal{T}_\ell$, $\ell > k$. Such a multilevel triangulation has nice structural properties. For example, the whole sequence $\mathcal{M}$ can be uniquely reconstructed from $\mathcal{T}_0$ and $\mathcal{T}_J$. A related property is that all the information on $\mathcal{M}$ can be represented in a natural way using a *hierarchical decomposition* $\mathcal{H} = (\mathcal{G}_0, \ldots, \mathcal{G}_J)$,

*Institut für Geometrie und Praktische Mathematik, RWTH Aachen, D-52056 Aachen, reusken@igpm.rwth-aachen.de, gross@igpm.rwth-aachen.de

$\mathcal{G}_0 := \mathcal{T}_0$, $\mathcal{G}_k := \mathcal{T}_k \setminus \mathcal{T}_{k-1},\ k \geq 1$. The hierarchical surplus on level k, $\mathcal{G}_k$, consists of all tetrahedra in $\mathcal{M}$ that are on the same level k. In the implementation of the refinement algorithm this hierarchical decomposition plays a key role.

Now assume that based on some error indicator certain tetrahedra in the finest triangulation $\mathcal{T}_J$ are marked for refinement. In many refinement algorithms one then modifies the finest triangulation $\mathcal{T}_J$ resulting in a new one, $\mathcal{T}_{J+1}$. Using such a strategy (which we call a *one*-level method) the new sequence $(\mathcal{T}_0, \ldots, \mathcal{T}_{J+1})$ is in general not a multilevel triangulation because the nestedness property 1 does not hold. We also note that when using such a method it is difficult to implement a reasonable coarsening strategy. In *multilevel* refinement algorithms the whole sequence $\mathcal{M}$ is used and as output one obtains a sequence $\mathcal{M}' = (\mathcal{T}'_0, \ldots, \mathcal{T}'_{J'})$, with $\mathcal{T}'_0 = \mathcal{T}_0$ and $J' \in \{J-1, J, J+1\}$. In general one has $\mathcal{T}'_k \neq \mathcal{T}_k$ for $k > 0$. We list a few important properties of this method:

- Both the input and output are *multilevel triangulations.*
- The method is *stable* and *consistent.*
- Local refinement and *coarsening* are treated in a similar way.
- The implementation uses only the hierarchical decomposition of $\mathcal{M}$. This allows *relatively simple data structures* without storage overhead.
- The costs are proportional to the number of tetrahedra in $\mathcal{T}_J$.

For a detailed discussion of these and other properties we refer to the literature ([6, 11, 12, 24]).

The multilevel structure of the refinement algorithm allows an efficient parallelization of this method. Such parallel multilevel refinement algorithms are known in the literature ([6, 24, 25]) and it has been shown that such a multilevel approach can have a high parallel efficiency even for very complex applications ([24, 7, 9, 10]). Clearly, a main problem related to the parallelization is how to store the distributed data on the different processors[1] of the distributed memory machine in a suitable way: One has to find a good compromise between storage overhead on the different processors and the costs of communication between the processors. The main topic of this paper is related to this issue. We will introduce a *new data partitioning format* that can be used in combination with a parallel multilevel refinement algorithm. Using this format, which is called an "admissible hierarchical decomposition", we can prove that the data partitioning between the processors is such that we have a favourable data locality (e.g., parent and children are on the same processor) and at the same time only a small amount of copies. We consider one particular variant of a parallel multilevel refinement algorithm. This method, that will be described in detail, is a parallel version of the serial algorithm discussed in [11]. We will prove that the application of the parallel refinement algorithm to an input admissible hierarchical decomposition yields an admissible hierarchical decomposition. This then proves that favourable data partitioning properties are preserved by the parallel refinement algorithm.
We do not know of any paper in which for a parallel multilevel refinement algorithm theoretical results related to important data partitioning properties (data locality, storage overhead) are presented.

The UG-package offers a very general parallel software-platform for solving CFD problems using multilevel mesh refinement with many different element types (triangles, quadrilaterals, tetrahedra, pyramids, prisms, hexahedra) in combination with

[1]here 'processor' means CPU with associated memory

parallel multigrid methods and dynamic load balancing (cf. [9]). In our analysis we are not able to treat such generality. We restrict ourselves to the theoretical analysis of a parallel multilevel refinement for tetrahedral meshes only. However, we believe that it is possible to derive similar results in a more general setting (eg., with other element types). The very important issue of load balancing is only briefly addressed in remark 9. We note that the data partitioning format that is introduced in this paper differs from but is similar to the one used in the UG-package (cf. remark 7). The parallel multilevel refinement algorithm that is treated in this paper is similar but not identical to the ones known in the literature (cf. remarks 4 and 7). We note, however, that a comparison with known methods is hard (or even impossible) due to the fact that usually only high-level versions of these methods are described in the literature (e.g. in [7]). These high-level descriptions are sufficient for explaining the principles underlying these methods. In the present paper, however, for the theoretical analysis a very precise description of the algorithm is necessary. Here we follow the presentation as in [11, 12], where theoretical properties of a *serial* multilevel refinement algorithm are derived.

This paper is organized as follows. In section 2 we collect some definitions. To make the paper more self-contained and in view of a better understanding of the parallel algorithm we decided to give a rather detailed description of the serial method (section 3). In section 4 we introduce and analyze the admissible hierarchical decomposition. Furthermore, the parallel multilevel refinement algorithm is presented. In section 4.1 we summarize the main new results of this paper. In section 5 a theoretical analysis is given which proves that the application of the parallel refinement algorithm to an input admissible hierarchical decomposition yields an admissible hierarchical decomposition. Finally, in section 6 we give results of a few numerical experiments.

2. Definitions and notation.

In this section we collect notations and definitions that will be used in the remainder of this paper. Let Ω be a polyhedral domain in $\mathbb{R}^3$.

Definition 1 (Triangulation). A finite collection $\mathcal{T}$ of tetrahedra $T \subset \overline{\Omega}$ is called a *triangulation* of Ω (or $\overline{\Omega}$) if the following holds:

1. $\mathrm{vol}(T) > 0$ for all $T \in \mathcal{T}$,
2. $\bigcup_{T\in\mathcal{T}} T = \overline{\Omega}$,
3. $\mathrm{int}(S) \cap \mathrm{int}(T) = \emptyset$ for all $S, T \in \mathcal{T}$ with $S \neq T$.

Definition 2 (Consistency). A triangulation $\mathcal{T}$ is called *consistent* if the intersection of any two tetrahedra in $\mathcal{T}$ is either empty, a common face, a common edge or a common vertex.

Definition 3 (Stability). A sequence of triangulations $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2, \ldots$ is called *stable* if all angles of all tetrahedra in this sequence are uniformly bounded away from zero.

It is known that for finite element discretizations in many cases the weaker (maximal angle) condition "all angles of all tetrahedra are uniformly bounded away from π" would be sufficient. However, using the latter condition, stronger requirements on the robustness of iterative solvers are needed, which can be avoided when using the minimal angle condition in definition 3.

Definition 4 (Refinement). For a given tetrahedron T a triangulation $\mathcal{K}(T)$ of T is called a *refinement* of T if $|\mathcal{K}(T)| \geq 2$ and any vertex of any tetrahedron $T' \in \mathcal{K}(T)$ is either a vertex or an edge midpoint of T. In this case T' is called a child of T and T is called the parent of T'. A refinement $\mathcal{K}(T)$ of T is called *regular* if $|\mathcal{K}(T)| = 8$, otherwise it is called *irregular*. A triangulation $\mathcal{T}_{k+1}$ is called *refinement* of a triangulation $\mathcal{T}_k \neq \mathcal{T}_{k+1}$ if for every $T \in \mathcal{T}_k$ either $T \in \mathcal{T}_{k+1}$ or $\mathcal{K}(T) \subset \mathcal{T}_{k+1}$ for some refinement $\mathcal{K}(T)$ of T.

DEFINITION 5 (Multilevel triangulation). A sequence of consistent triangulations $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ is called a *multilevel triangulation* of Ω if the following holds:

1. For $0 \le k < J$: $\mathcal{T}_{k+1}$ is a refinement of $\mathcal{T}_k$.
2. For $0 \le k < J$: $T \in \mathcal{T}_k \cap \mathcal{T}_{k+1} \Rightarrow T \in \mathcal{T}_J$.

The tetrahedra $T \in \mathcal{T}_J$ are called the leaves of $\mathcal{M}$. Note that T is a leaf iff T has no children in $\mathcal{M}$. A tetrahedron $T \in \mathcal{M}$ is called *regular* if $T \in \mathcal{T}_0$ or T resulted from a regular refinement of its parent. Otherwise T is called *irregular*. A multilevel triangulation $\mathcal{M}$ is called regular if all irregular $T \in \mathcal{M}$ are leaves (i.e., have no children in $\mathcal{M}$).

REMARK 1. Let $\mathcal{M}$ be a multilevel triangulation and V_k $(0 \le k \le J)$ be the corresponding finite element spaces of continuous functions $p \in C(\bar{\Omega})$ such that $p_{|T} \in \mathcal{P}_q$ for all $T \in \mathcal{T}_k$ $(q \ge 1)$. The refinement property 1 in definition 5 implies nestedness of these finite element spaces: $V_k \subset V_{k+1}$.

DEFINITION 6 (Hierarchical decomposition of $\mathcal{M}$). Let $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ be a multilevel triangulation of Ω. For every tetrahedron $T \in \mathcal{M}$ a unique level number $\ell(T)$ is defined by $\ell(T) := \min\{\, k \mid T \in \mathcal{T}_k \,\}$. The set $\mathcal{G}_k \subset \mathcal{T}_k$

$$\mathcal{G}_k := \{\, T \in \mathcal{T}_k \mid \ell(T) = k \,\}$$

is called the *hierarchical surplus* on level k $(k = 0, 1, \ldots, J)$. Note that $\mathcal{G}_0 = \mathcal{T}_0$, $\mathcal{G}_k = \mathcal{T}_k \setminus \mathcal{T}_{k-1}$ for $k = 1, \ldots, J$. The sequence $\mathcal{H} = (\mathcal{G}_0, \ldots, \mathcal{G}_J)$ is called the *hierarchical decomposition* of $\mathcal{M}$. Note that the multilevel triangulation $\mathcal{M}$ can be reconstructed from its hierarchical decomposition.

REMARK 2. The hierarchical decomposition induces simple data structures in a canonical way. The tetrahedra of each hierarchical surplus $\mathcal{G}_k$ are stored in a separate list. Thus every tetrahedron $T \in \mathcal{M}$ is stored exactly once since T has a unique level number $\ell(T)$. By introducing unique level numbers also for vertices, edges and faces, these subsimplices can be stored in the same manner: For a subsimplex S the level number $\ell(S)$ is defined as the level of its first appearance. Additionally, the objects are linked to certain corresponding objects by pointers (e.g., a tetrahedron is linked to its vertices, edges, faces, children and parent).

3. A serial multilevel refinement algorithm.

In this section we describe a refinement algorithm which is, apart from some minor modifications, the algorithm presented in [11, 12]. This method is based on similar ideas as the refinement algorithms in [4, 6, 7]. At the end of this section (remark 4) we discuss the differences between the method presented here and the one from [11, 12]. We give a rather detailed description of the algorithm to facilitate the presentation of the *parallel* version of this method that is given in section 4.3. The refinement strategy is based on a set of regular and irregular refinement rules (also called red and green rules, due to [2, 4, 3, 5]). The regular and irregular rules are local in the sense that they are applied to a single tetrahedron. These rules are applied in a (global) refinement algorithm (section 3.3) that describes how the local rules can be combined to ensure consistency and stability.

3.1. The regular refinement rule.

Let T be a given tetrahedron. For the construction of a regular refinement of T it is natural to connect midpoints of the edges of T by subdiving each of the faces into four congruent triangles. This yields four subtetrahedra at the corners of T (all similar to T) and an octahedron in the middle. For the subdivision of this octahedron into four subtetrahedra with equal volume there are three possibilities corresponding to the three diagonals that connect opposite vertices of the octahedron. One has to be careful in choosing an appropriate subdivision since in [35] it has been shown that the wrong choice may lead to a sequence of triangulations that is not stable. A stable tetrahedral regular refinement strategy, based on an idea from [17], is presented in [11, 13]. We recall this method.

Let $T = [x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}]$ be a tetrahedron with *ordered* vertices $x^{(1)}$, $x^{(2)}$, $x^{(3)}$, $x^{(4)}$ and

$$x^{(ij)} := \frac{1}{2}(x^{(i)} + x^{(j)}) \ , \quad 1 \le i < j \le 4,$$

the midpoint of the edge between $x^{(i)}$ and $x^{(j)}$. The regular refinement $\mathcal{K}(T) := \{T_1, \ldots, T_8\}$ of T is constructed by the (red) rule

$$\begin{aligned} T_1 &:= [x^{(1)}, x^{(12)}, x^{(13)}, x^{(14)}] \ , & T_5 &:= [x^{(12)}, x^{(13)}, x^{(14)}, x^{(24)}] \ , \\ T_2 &:= [x^{(12)}, x^{(2)}, x^{(23)}, x^{(24)}] \ , & T_6 &:= [x^{(12)}, x^{(13)}, x^{(23)}, x^{(24)}] \ , \\ T_3 &:= [x^{(13)}, x^{(23)}, x^{(3)}, x^{(34)}] \ , & T_7 &:= [x^{(13)}, x^{(14)}, x^{(24)}, x^{(34)}] \ , \\ T_4 &:= [x^{(14)}, x^{(24)}, x^{(34)}, x^{(4)}] \ , & T_8 &:= [x^{(13)}, x^{(23)}, x^{(24)}, x^{(34)}] \ . \end{aligned} \tag{3.1}$$

In [13] it is shown that for any T the repeated application of this rule produces a sequence of consistent triangulations of T which is stable. For a given T all tetrahedra that are generated in such a recursive refinement process form at most three similarity classes.

3.2. Irregular refinement rules. Let $\mathcal{T}$ be a given consistent triangulation. We select a subset $\mathcal{S}$ of tetrahedra from $\mathcal{T}$ and assume that the regular refinement rule is applied to each of the tetrahedra from $\mathcal{S}$. In general the resulting triangulation $\mathcal{T}'$ will not be consistent. The irregular (or green) rules are used to make this new triangulation consistent. For this we introduce the notions of an edge counter and edge refinement pattern. The *edge counter* $C(E)$, that depends on $\mathcal{T}$ and $\mathcal{S}$, assigns an integer value to each of the edges E of $\mathcal{T}$ as follows: $C(E) = m$ if the edge E is an edge of precisely m elements from $\mathcal{S}$. Hence, the edge E has been refined when going from $\mathcal{T}$ to $\mathcal{T}'$ iff $C(E) > 0$ holds. Related to this, for $T \in \mathcal{T}$ we define the *edge refinement pattern* $R(T)$ as follows. Let $E_1, \ldots, E_6$ be the ordered edges of T. We define the 6-tuple

$$R = (r_1, \ldots, r_6) \in \{0, 1\}^6$$

by: $r_i = 0$ if $C(E_i) = 0$ and $r_i = 1$ if $C(E_i) > 0$. This edge refinement pattern describes a local property in the sense that it is related to a single tetrahedron and its value directly follows from the values of the edge counter. For $T \in \mathcal{S}$ we have $R(T) = (1, \ldots, 1)$. For $T \in \mathcal{T} \setminus \mathcal{S}$ the case $R(T) = (0, \ldots, 0)$ corresponds to the situation that the tetrahedron T does not contain any vertices from $\mathcal{T}'$ at the midpoints of its edges. For each of the $2^6 - 1$ possible patterns $R \neq (0, \ldots, 0)$ there exists a corresponding refinement $\mathcal{K}(T)$ of T (in the fashion of (3.1)) for which the vertices of the children coincide with vertices of T or with the vertices at the midpoints on the edges E_i with $r_i = 1$. This refinement, however, is not always unique. This is illustrated in figure 3.1.

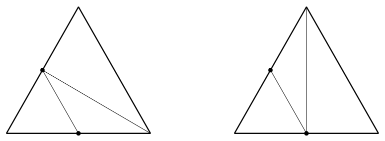


FIG. 3.1. *Non-unique face refinement*

To obtain a consistent triangulation in which the subdivision of adjacent faces matches special care is needed. One way to ensure consistency is by introducing a so-called consistent vertex numbering:

DEFINITION 7 (Consistent vertex numbering). Let T_1 and T_2 be two adjacent tetrahedra with a common face $F = T_1 \cap T_2$ and local vertex ordering

$$T_l = [x_l^{(1)}, x_l^{(2)}, x_l^{(3)}, x_l^{(4)}], \qquad l = 1, 2.$$

Let the vertex set of F be given by

$$\{x_1^{(i_1)}, x_1^{(i_2)}, x_1^{(i_3)}\} = \{x_2^{(j_1)}, x_2^{(j_2)}, x_2^{(j_3)}\} \ .$$

Without loss of generality we assume $i_1 < i_2 < i_3$ and $j_1 < j_2 < j_3$. The pair (T_1, T_2) has a *consistent vertex numbering* if

$$x_1^{(i_k)} = x_2^{(j_k)}, \quad k = 1, 2, 3$$

holds, i.e., if the ordering of the vertices of F induced by the vertex numbering of T_1 coincides with the one induced by the vertex numbering of T_2. A consistent triangulation $\mathcal{T}$ has a *consistent vertex numbering* if every two neighboring tetrahedra have this property.

We note that a consistent vertex numbering can be constructed in a rather simple way. Consider an (initial) triangulation $\tilde{\mathcal{T}}$ with an arbitrary numbering of its vertices. This global numbering induces a canonical local vertex ordering which is a consistent vertex numbering of $\tilde{\mathcal{T}}$. Furthermore, each refinement rule can be defined such that the consistent vertex numbering property of the parent is inherited by its children by prescribing suitable local vertex orderings of the children. (3.1) is an example of such a rule. Using such a strategy a consistent triangulation $\tilde{\mathcal{T}}'$ that is obtained by refinement of $\tilde{\mathcal{T}}$ according to these rules also has a consistent vertex numbering.

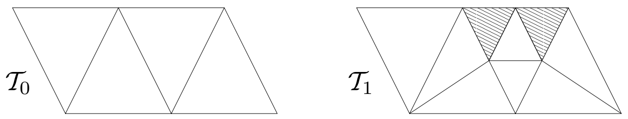
ASSUMPTION 1. *In the remainder of this paper we always assume that the initial triangalution $\mathcal{T}_0$ is consistent and has a consistent vertex numbering.*

Assume that the given triangulation $\mathcal{T}$ has a consistent vertex numbering. For a face with a pattern as in figure 3.1 one can then define a unique face refinement by connecting the vertex with the smallest number with the midpoint of the opposite edge. *For each edge refinement pattern $R \in \{0,1\}^6$ we then have a unique rule.* We emphasize that if the edge refinement pattern is known *for the application of the regular or irregular rules to a given tetrahedron no information from neighboring tetrahedra is needed.* Clearly, for parallelization this is a very nice property.

Up to now we discussed how the *consistency* of a triangulation can be achieved by the choice of suitable irregular refinement rules based on the consistent vertex numbering property. We will now explain how the regular and irregular rules can be combined in a repeated refinement procedure to obtain a *stable* sequence of consistent triangulations. The crucial point is to *allow only the refinement of regular tetrahedra,* i.e. children of irregularly refined tetrahedra, also called *green children*, are never refined. If such a green child T is marked for refinement, instead of refining T the irregular refinement of the parent will be replaced by a regular one. As the application of the regular rule (3.1) creates tetrahedra of at most 3 similarity classes (cf. [17, 13]), the tetrahedra created by a refinement procedure according to this strategy belong to an a-priori bounded number of similarity classes. Hence the obtained sequence of triangulations is stable.

3.3. The multilevel refinement algorithm. In this section we describe how the regular and irregular refinement rules are used in a multilevel refinement algorithm (as in [18, 11, 12]).

We first illustrate the difference to one-level refinement methods. For ease of presentation, we use triangles instead of tetrahedra in our examples. In one-level methods (as in [2, 4, 3, 5]) there is a loop over all triangles in the *finest* triangulation $\mathcal{T}_J$. Each triangle is processed, based on a (small) number of (red and green) refinement rules. If necessary (e.g., to maintain consistency), neighboring triangles are refined, too. In general the family of triangulations $\mathcal{T}_0, \ldots, \mathcal{T}_J$ that is constructed will not be nested. We emphasize that, although in the implementation of such one-level methods a multilevel (tree) structure may be used, there is no loop over the different triangulations $\mathcal{T}_0, \ldots, \mathcal{T}_J$ in the family. As a simple example, consider the following multilevel triangulation $(\mathcal{T}_0, \mathcal{T}_1)$:



In $\mathcal{T}_1$ two triangles are marked (by shading) for further refinement. The one-level method from [4] uses the finest triangulation $\mathcal{T}_1$ as input and applies a regular refinement rule followed by irregular refinement rules:

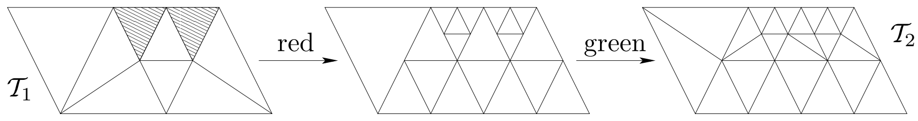


Fig. 3.2. *One-level red/green refinement*

As output one obtains a consistent triangulation $\mathcal{T}_2$ which is *not* a refinement of $\mathcal{T}_1$ (in the sense of definition 4). Related to this we note that the finite element spaces corresponding to $\mathcal{T}_1, \mathcal{T}_2$ are not nested (cf. remark 1) and that it is not obvious how to construct a hierarchical decomposition of the sequence $\mathcal{T}_0, \mathcal{T}_1, \mathcal{T}_2$. These are disadvantages if one uses such grid refinement techniques in combination with multigrid solvers for the numerical solution of partial differential equations. An advantage of the one-level method compared to the multilevel strategy discussed below is its simplicity. In multilevel refinement algorithms both the input and the output are multilevel triangulations (definition 5). In the implementation only the hierarchical decompositions of these multilevel triangulations are used.

We now introduce the notions of `status` and `mark` of a tetrahedron that will be used in the subroutines of the multilevel refinement algorithm. Let $\mathcal{M} = (\mathcal{T}_0, \ldots, \mathcal{T}_J)$ be a multilevel triangulation that has been constructed by applying the regular and irregular refinement rules and let $\mathcal{H} = (\mathcal{G}_0, \ldots, \mathcal{G}_J)$ be the corresponding hierarchical decomposition. Every tetrahedron $T \in \mathcal{H}$ is either a leaf of $\mathcal{M}$ (i.e., $T \in \mathcal{T}_J$) or it has been refined. The label status is used to describe this property of T:

$$\text{For} \quad T \in \mathcal{H}: \quad \mathsf{status}(T) = \begin{cases} \mathsf{NoRef} & \text{if} \quad T \quad \text{is a leaf of } \mathcal{M} \\ \mathsf{RegRef} & \text{if} \quad T \quad \text{is regularly refined in } \mathcal{M} \\ \mathsf{IrregRef} & \text{if} \quad T \quad \text{is irregularly refined in } \mathcal{M} \end{cases}$$

We note that the label $\mathsf{IrregRef}$ contains the number of the irregular refinement rule (one out of 63) that has been used to refine T, i.e., the binary representation of $\mathsf{status}(T)$ coincides with the edge refinement pattern of T.

In adaptive refinement an error estimator (or indicator) is used to mark certain elements of $\mathcal{T}_J$ for further refinement or for deletion. For this the label mark is used:

$$\text{For} \quad T \in \mathcal{H}: \quad \mathsf{mark}(T) = \begin{cases} \mathsf{Ref} & \text{if} \quad T \in \mathcal{T}_J \quad \text{is marked for refinement} \\ \mathsf{Del} & \text{if} \quad T \in \mathcal{T}_J \quad \text{is marked for deletion} \\ \mathsf{status}(T) & \text{otherwise} \end{cases}$$

We describe a multilevel refinement algorithm known in the literature. The basic form of this method was introduced by Bastian [6] and developed further in the UG-group [7, 9, 24, 25, 33]. We use the presentation as in [11, 12], which is shown in figure 3.3.

The input of *SerRefinement* consists of a hierarchical decomposition $\mathcal{G}_0, \ldots, \mathcal{G}_J$ in

```
Algorithm SerRefinement($\mathcal{G}_0, \ldots, \mathcal{G}_J$)
    for $k = J, \ldots, 0$ do                                  // phase I
        DetermineMarks($\mathcal{G}_k$);                              (1)
        MarksForClosure($\mathcal{G}_k$);                             (2)
    for $k = 0, \ldots, J$ do if $\mathcal{G}_k \neq \emptyset$ then                // phase II
        if $k > 0$ then MarksForClosure($\mathcal{G}_k$);             (3)
        if $k < J$ then Unrefine($\mathcal{G}_k$);                    (4)
        Refine($\mathcal{G}_k$);                                      (5)
    if $\mathcal{G}_J = \emptyset$ then $J := J - 1$;                           (6)
    else if $\mathcal{G}_{J+1} \neq \emptyset$ then $J := J + 1$;                 (7)
```

Fig. 3.3. *Serial multilevel refinement algorithm.*

which all refined tetrahedra T are labeled by $\mathsf{mark}(T) = \mathsf{status}(T)$ according to their status and the unrefined $T \in \mathcal{T}_J$ have $\mathsf{mark}(T) \in \{\mathsf{NoRef}, \mathsf{Ref}, \mathsf{Del}\}$. The output is again a hierarchical decomposition, where all tetrahedra are marked according to their status.
The main idea underlying the algorithm *SerRefinement* is illustrated using the multilevel triangulation $(\mathcal{T}_0, \mathcal{T}_1)$ from above. The hierarchical decomposition and the corresponding marks are shown in figure 3.4.

Note that for the two shaded triangles in $\mathcal{G}_1$ we have $\mathsf{status}(T) \neq \mathsf{mark}(T)$. For all other triangles $\mathsf{status}(T) = \mathsf{mark}(T)$ holds. In phase I of the algorithm (top–down: (1),(2)) only marks are changed.

Once phase I has been completed the marks have been changed such that $\mathsf{mark}(T) \in \{\mathsf{NoRef}, \mathsf{RegRef}, \mathsf{IrregRef}\}$ holds for all $T \in \mathcal{H}$. We emphasize that all green children in $\tilde{\mathcal{G}}_1$ have $\mathsf{mark}(T) = \mathsf{NoRef}$, as they are not refined because of stability reasons. Instead

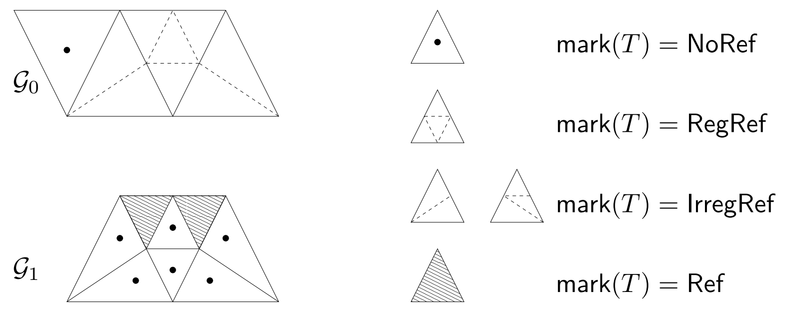

$\mathcal{G}_0$
$\mathcal{G}_1$
$\mathsf{mark}(T) = \mathsf{NoRef}$
$\mathsf{mark}(T) = \mathsf{RegRef}$
$\mathsf{mark}(T) = \mathsf{IrregRef}$
$\mathsf{mark}(T) = \mathsf{Ref}$

FIG. 3.4. *Input hierarchical decomposition.*

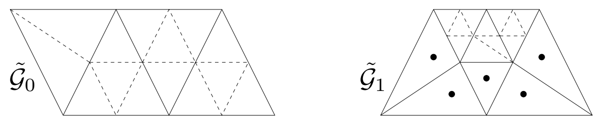


FIG. 3.5. *After phase I.*

the corresponding irregular refined parents in $\tilde{\mathcal{G}}_0$ are labeled by $\mathsf{mark}(T) = \mathsf{RegRef}$. In the second phase (bottom–up: (3)-(5)) the actual refinement (unrefinement is not needed in our example) is constructed:

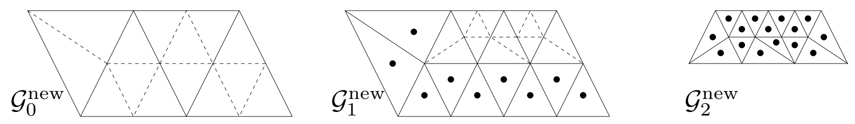

$\mathcal{G}_0^{\text{new}}$
$\mathcal{G}_1^{\text{new}}$
$\mathcal{G}_2^{\text{new}}$

FIG. 3.6. *Output hierarchical decomposition.*

In the output hierarchical decomposition $\mathcal{H} = (\mathcal{G}_0^{\text{new}}, \mathcal{G}_1^{\text{new}}, \mathcal{G}_2^{\text{new}})$ we have $\mathsf{mark}(T) = \mathsf{status}(T)$ for all $T \in \mathcal{H}$. The output multilevel triangulation $\mathcal{M} = (\mathcal{T}_0^{\text{new}}, \mathcal{T}_1^{\text{new}}, \mathcal{T}_2^{\text{new}})$ is *regular* (cf. definition 5) and is given by

$$\mathcal{T}_0^{\text{new}} = \mathcal{G}_0^{\text{new}}, \quad \mathcal{T}_1^{\text{new}} = \mathcal{G}_1^{\text{new}}, \quad \mathcal{T}_2^{\text{new}} = \mathcal{G}_2^{\text{new}} \cup \{\, T \in \mathcal{G}_1^{\text{new}} \mid \mathsf{mark}(T) = \mathsf{NoRef}\,\} \; .$$

Note that $\mathcal{T}_0^{\text{new}} = \mathcal{T}_0$, $\mathcal{T}_1^{\text{new}} \neq \mathcal{T}_1$ (!) and that the new finest triangulation $\mathcal{T}_2^{\text{new}}$ is the same as the triangulation $\mathcal{T}_2$ in figure 3.2 resulting from the one-level algorithm.

Below we describe the subroutines used in algorithm *SerRefinement*. A detailed discussion of these subroutines is given in [11, 12].

DetermineMarks. In this subroutine only marks are changed. The new values of the marks that are changed are of the type RegRef or NoRef. The value $\mathsf{mark}(T) = \mathsf{RegRef}$ is assigned if

- T is a regular leaf with $\mathsf{mark}(T) = \mathsf{Ref}$
- T has been irregularly refined ($\mathsf{status}(T) = \mathsf{IrregRef}$) and at least one of its children is marked for refinement (either by $\mathsf{mark}(\text{child}) = \mathsf{Ref}$ or by a certain edge refinement pattern)

The value $\mathsf{mark}(T) = \mathsf{NoRef}$ is assigned if

- $\mathsf{status}(T) = \mathsf{RegRef}$ and all children of T are marked for deletion ($\mathsf{mark}(T') = \mathsf{Del}$ for all $T' \in \mathcal{K}(T)$)
- T has been irregularly refined ($\mathsf{status}(T) = \mathsf{IrregRef}$) and none of its children is marked for refinement

The subroutine is described in detail in figure 3.7. Another important task of *De-*

```
Function DetermineMarks($\mathcal{G}_k$)
    for $T \in \mathcal{G}_k$ do
        if status($T$) = NoRef then
            if $T$ is regular and mark($T$) = Ref then
                mark($T$) := RegRef;                                      (1)
                increase edge counters;                                   (2)
            if $k = 0$ and mark($T$) = Del then mark($T$) := NoRef;       (3)
        else if status($T$) = RegRef then
            if $\forall T' \in \mathcal{K}(T)$ : mark($T'$) = Del then
                mark($T$) := NoRef;                                       (4)
                decrease edge counters;                                   (5)
            for $T' \in \mathcal{K}(T)$ do
                if mark($T'$) = Del then mark($T'$) := NoRef;             (6)
        else // status($T$) = IrregRef
            if $\exists T' \in \mathcal{K}(T)$ : mark($T'$) = Ref
            or an edge of a child, which is not an edge of $T$,
                is marked for refinement then
                mark($T$) := RegRef;                                      (7)
                increase edge counters;                                   (8)
            else mark($T$) := NoRef;                                      (9)
            for $T' \in \mathcal{K}(T)$ do mark($T'$) := NoRef;           (10)
```

FIG. 3.7. *Subroutine DetermineMarks.*

termineMarks is the book-keeping of the edge counters. If a tetrahedron is to be regularly refined the counters of its edges are increased. Similarly, if all children of a regularly refined tetrahedron are removed the edge counters of its edges are decreased.

MarksForClosure. In this subroutine an appropriate irregular refinement rule is determined for an element $T \in \mathcal{G}_k$ to avoid hanging nodes. The tetrahedron T must be regular (irregular elements are never refined) and it should not be marked for regular refinement. The subroutine is described in figure 3.8.

```
Function MarksForClosure($\mathcal{G}_k$)
    for $T \in \mathcal{G}_k$ do
        if $T$ is regular and mark($T$) $\neq$ RegRef then
            Determine the edge refinement pattern $R$
            of $T$ (using the edge counters);                             (1)
            if $R = (0, \ldots, 0)$ and mark($T$) = Del then
                do nothing;
                // NoRef-mark is set in DetermineMarks($\mathcal{G}_{k-1}$)
            else
                mark($T$) := $R$;                                         (2)
```

FIG. 3.8. *Subroutine MarksForClosure.*

Unrefine. In the call of this subroutine on level k all the tetrahedra, vertices, edges and faces on level $k+1$ that are not needed anymore (due to changed marks) are removed. More details are given in figure 3.9. We note that for an efficient implementation one could check also for the case status$(T) \neq$ NoRef and mark$(T) \neq$ status(T) whether certain already known objects (tetrahedra, edges, etc.) on level $k+1$ can be reused in the refinement.

```
Function Unrefine($\mathcal{G}_k$)
    Label all tetrahedra, vertices, edges and faces on level $k+1$ for deletion;    (1)
    for $T \in \mathcal{G}_k$ do if (status($T$) $\neq$ NoRef and mark($T$) = status($T$)) then
        Remove all deletion labels of the
        children of $T$ and of their vertices, edges, faces;    (2)
    Remove all tetrahedra, vertices, edges and faces on level $k+1$
    that are labeled for deletion;    (3)
```

FIG. 3.9. *Subroutine Unrefine.*

Refine. In the subroutine *Refine*, if mark(T) $\neq$ status(T), an appropriate refinement of T, based on mark(T), is made and new objects (tetrahedra, vertices, edges and faces) are created on level $k+1$. For $k = J$, a new hierarchical surplus $\mathcal{G}_{J+1}$ is constructed. The refined tetrahedron T is labeled with its new status. After application of the subroutine *Refine* on level k all tetrahedra on this level have a status which corresponds to their mark. Leaves are given the status NoRef. Further details are presented in figure 3.10.

```
Function Refine($\mathcal{G}_k$)
    if $k = J$ then $\mathcal{G}_{k+1} := \emptyset$;    (1)
    for $T \in \mathcal{G}_k$ do
        if mark($T$) $\neq$ status($T$) then
            Refine $T$ according to mark($T$);    (2)
            status($T$) := mark($T$);    (3)
            for $T' \in \mathcal{K}(T)$ do
                Find existing vertices, edges and faces of $T'$;    (4)
                create missing vertices, edges and faces of $T'$;    (5)
                status($T'$) := NoRef;    (6)
```

FIG. 3.10. *Subroutine Refine.*

REMARK 3. When applying the subroutines to $\mathcal{G}_k$, for any $T \in \mathcal{G}_k$ one only needs information that is directly connected with T (e.g., mark(T)), information about a possible parent (to decide, for example, whether T is regular or not) and if T is not a leaf, information about the children of T.

REMARK 4. As already noted above, algorithm *SerRefinement* is very similar to the method of BEY ([11, 12]). There are, however, some differences. We use a complete set of 64 refinement rules, whereas BEY uses a much smaller number of green rules, which may cause a domino effect. In view of data locality and parallelization we wanted to avoid this effect. Concerning the implementation there are two major differences to the approach of BEY. Firstly, in our implementation faces are explicitly represented in the data structure, which simplifies the access from one tetrahedron to its four neighbor tetrahedra. Secondly, each vertex, edge and face is stored only once (and not on every level where corresponding tetrahedra occur).
The algorithm used in UG [7] (cf. [6, 7, 24]) is of the same V-cycle structure but more general as it can deal with other element types such as hexahedra, prisms and pyramids, too. A complete set of refinement rules is used for each element type. Faces are not represented, which saves memory on the one hand, but on the other hand introduces so called horizontal ghosts in the parallel data structure (cf. remark 7 below).

4. Parallelization of the multilevel refinement method.

4.1. Introduction. In many applications (for example, CFD) the computational complexity is very high and one wants to use parallel machines. Here we only consider distributed memory machines which use message passing (MPI). In this section we reconsider the local refinement algorithm *SerRefinement* and present a version, which is called *ParRefinement*, that is suitable for such a machine. For a given input-multilevel triangulation the parallel method produces the same output-multilevel triangulation as the serial method. In this sense the "computational part" of the algorithm is not changed. In the parallel case load has to be distributed uniformly among the processors, so in practice an adaptive parallel refinement algorithm is combined with dynamic load balancing and data migration between the processors. One main issue related to the parallelization of the algorithm *SerRefinement* is to find an appropriate distributed storage of data with regard to data locality. Note that an *overlapping distribution of the elements is necessary*, due to the fact that parents and children are linked by pointers. Consider, for example, the situation in which a parent T and its child T' are stored on different processors, say p and q. Since pointers from one local memory to another are not allowed in a distributed memory setting, we have to use a copy to realize this pointer. One could store a copy of T on processor q to represent the link between T and T' as a pointer on processor q. If one does not allow such ghost copies, all ancestors and descendants of a tetrahedron must be on the same processor. This would cause very coarse data granularity, poor load balancing and hence low parallel efficiency.

One main topic of this paper is the introduction and analysis of a new data distribution format that is very suitable for parallelization of the multilevel refinement algorithm. This data distribution format is such that the following holds:

1. Let $T \in \mathcal{G}_k$ be an element from the hierarchical surplus on level k. Then T is stored on one processor, say p, as a so called master element. In certain cases (explained below) a ghost copy of T is stored on *one* other processor, say q.
2. The children of T (if they exist) are all stored as masters either on processor p or, if T has a ghost copy, on processor q. For $T \in \mathcal{G}_k$, $k > 0$, the parent of T or a copy of it is stored on processor p.

In multilevel refinement a crucial point is that for a tetrahedron T one needs information about all children of T. Due to property 2 this information is available on the local processor (p or q) *without communication*. The first property shows that in a certain sense the overlap of tetrahedra is small.

The new data distribution format will be made mathematically precise by a formal specification of a so-called *admissible hierarchical decomposition*. This is presented in section 4.2. In section 4.3 we introduce the *parallel* version of the multilevel refinement algorithm (*ParRefinement*).

The main results concerning the admissible hierarchical decomposition and the parallel multilevel refinement method can be summarized as follows:

a. An admissible hierarchical decomposition has the desirable properties 1 (small storage overhead) and 2 (data locality) from above. This is proved in section 4.2.
b. The application of the algorithm *ParRefinement* to an admissible hierarchical decomposition results in an admissible hierarchical decomposition. This is proved in section 5.
c. Given an admissible hierarchical decomposition one can apply a suitable load balancing and data migration algorithm such that after data migration one still has an admissible hierarchical decomposition. We comment on this in

remark 9 below.

4.2. Admissible hierarchical decomposition. Let the sequence $\mathcal{M} = (\mathcal{T}_0, \dots, \mathcal{T}_J)$ of triangulations be a multilevel triangulation and $\mathcal{H} = (\mathcal{G}_0, \dots, \mathcal{G}_J)$ the corresponding hierarchical decomposition. In this section we introduce a particular format for the distribution of the tetrahedra in $\mathcal{H}$ among processors on a parallel machine. We assume that the processors are numbered $1, \dots, P$.

For the set of elements in the hierarchical surplus on level k that are stored on processor p we introduce the notation

$$\mathcal{G}_k(p) := \{\, T \in \mathcal{G}_k \mid T \text{ is stored on processor } p \,\}$$

and we define

$$\mathcal{H}(p) := (\mathcal{G}_0(p), \dots, \mathcal{G}_J(p)) \ .$$

In general the intersection $\mathcal{G}_k(p) \cap \mathcal{G}_k(q)$, $p \neq q$, may be nonempty. Also note that in general $\mathcal{H}(p)$ is not a hierarchical decomposition (in the sense of definition 6). The sequence

$$\tilde{\mathcal{H}} = (\mathcal{H}(1), \dots, \mathcal{H}(P)) \tag{4.1}$$

is called a *distributed hierarchical decomposition* (corresponding to $\mathcal{H}$).

For each level k and processor p we introduce a set of *master elements*, $\mathsf{Ma}_k(p) \subset \mathcal{G}_k(p)$, and a set of *ghost elements*, $\mathsf{Gh}_k(p) \subset \mathcal{G}_k(p)$. In the formulation of the conditions below we use: $\mathcal{K}(T) := \emptyset$ if $\mathsf{status}(T) = \mathsf{NoRef}$, and $\mathsf{Ma}_{J+1}(p) := \emptyset$.

We now formalize the conditions on data distribution as follows.

Definition 8 (Admissible hierarchical decomposition). The distributed hierarchical decomposition $\tilde{\mathcal{H}}$ is called an *admissible hierarchical decomposition* if for all $k = 1, \dots, J$ the following conditions are fulfilled:

(A1) **Partitioning of $\mathcal{G}_k(p)$:** The sets of masters and ghosts form a disjoint partitioning of $\mathcal{G}_k(p)$:

$$\forall p \quad \mathsf{Ma}_k(p) \cup \mathsf{Gh}_k(p) = \mathcal{G}_k(p) \quad \text{and} \quad \mathsf{Ma}_k(p) \cap \mathsf{Gh}_k(p) = \emptyset$$

(A2) **Existence:** Every element from $\mathcal{G}_k$ is represented as a master element on level k:

$$\mathcal{G}_k = \bigcup_{p=1}^{P} \mathsf{Ma}_k(p)$$

(A3) **Uniqueness:** Every element from $\mathcal{G}_k$ is represented by at most one master element on level k:

$$\forall p_1, p_2 : \qquad \mathsf{Ma}_k(p_1) \cap \mathsf{Ma}_k(p_2) \neq \emptyset \;\Rightarrow\; p_1 = p_2$$

(A4) **Child–parent locality:** A child master element and its parent (as master or ghost) are stored on the same processor:

$$\forall p \quad \forall T \in \mathcal{G}_k \quad \forall T' \in \mathcal{K}(T) : \qquad T' \in \mathsf{Ma}_{k+1}(p) \;\Rightarrow\; T \in \mathcal{G}_k(p)$$

(A5) **Ghosts are parents:** Ghost elements always have children:

$$\forall p \quad \forall T \in \mathsf{Gh}_k(p) : \qquad \mathcal{K}(T) \neq \emptyset$$

(A6) **Ghost–children locality:** A ghost element and its children are stored on the same processor:

$$\forall p \quad \forall T \in \mathsf{Gh}_k(p): \qquad \mathcal{K}(T) \subset \mathsf{Ma}_{k+1}(p)$$

REMARK 5. Consider a consistent initial triangulation $\mathcal{T}_0 = \mathcal{G}_0$ with a nonoverlapping distribution of the tetrahedra: $\mathcal{G}_0(p) \cap \mathcal{G}_0(q) = \emptyset$ for all $p \neq q$. In this case all tetrahedra can be stored as masters and there are no ghosts. Then the distributed hierarchical decomposition $\tilde{\mathcal{H}} = ((\mathcal{G}_0(1)), \ldots, (\mathcal{G}_0(P)))$ is admissible.

Two elementary results are given in:

LEMMA 4.1. *Let $\tilde{\mathcal{H}}$ as in (4.1) be a distributed hierarchical decomposition. The following holds:*

1. *If the conditions (A3), (A5) and (A6) are satisfied then for any element from $\mathcal{G}_k$ there is at most one corresponding ghost element:*

$$\forall T \in \mathcal{G}_k \quad \forall p, q: \qquad T \in \mathsf{Gh}_k(p) \cap \mathsf{Gh}_k(q) \ \Rightarrow\ p = q$$

2. *If the conditions (A1), (A2), (A3), (A4) and (A6) are satisfied then all children of a parent are stored as master elements on one processor:*

$$\forall T \in \mathcal{G}_k \quad \exists p: \qquad \mathcal{K}(T) \subset \mathsf{Ma}_{k+1}(p)$$

Proof.

1. Take $T \in \mathcal{G}_k$ and p, q such that $T \in \mathsf{Gh}_k(p) \cap \mathsf{Gh}_k(q)$. Then because of (A5) and (A6) we obtain $\emptyset \neq \mathcal{K}(T) \subset \mathsf{Ma}_{k+1}(p) \cap \mathsf{Ma}_{k+1}(q)$. From (A3) we conclude $p = q$.
2. Take $T \in \mathcal{G}_k$. For $\mathcal{K}(T) = \emptyset$ nothing has to be proven, so we consider $\mathcal{K}(T) \neq \emptyset$. Choose $T' \in \mathcal{K}(T) \subset \mathcal{G}_{k+1}$. Then because of (A2)

$$\exists p: \qquad T' \in \mathsf{Ma}_{k+1}(p) \ ,$$

and due to (A4) T is also stored on processor p: $T \in \mathcal{G}_k(p)$. From (A1) it follows that T is stored either as a ghost element or a master element on processor p. First consider $T \in \mathsf{Gh}_k(p)$. Then (A6) implies $\mathcal{K}(T) \subset \mathsf{Ma}_{k+1}(p)$, *qed.*
Now consider the case $T \in \mathsf{Ma}_k(p)$. Suppose another child $T'' \in \mathcal{K}(T)$, $T'' \neq T'$, is *not* stored as master element on processor p, i.e. $T'' \notin \mathsf{Ma}_{k+1}(p)$. From (A2) it follows that there is a processor q, $q \neq p$, with $T'' \in \mathsf{Ma}_{k+1}(q)$, and (A4) yields $T \in \mathcal{G}_k(q)$. From $T \in \mathsf{Ma}_k(p)$, (A3) and (A1) we obtain $T \in \mathsf{Gh}_k(q)$ and hence (A6) yields $\mathcal{K}(T) \subset \mathsf{Ma}_{k+1}(q)$. In particular we have $T' \in \mathsf{Ma}_{k+1}(q)$ and thus

$$T' \in \mathsf{Ma}_{k+1}(p) \cap \mathsf{Ma}_{k+1}(q) \ ,$$

with $p \neq q$. This is a contradiction to (A3). Hence for all children $T'' \in \mathcal{K}(T)$ $T'' \in \mathsf{Ma}_{k+1}(p)$ holds.

∎

REMARK 6. Due to the conditions (A2) and (A3) every tetrahedron $T \in \mathcal{H}$ can be assigned a unique processor on which T is stored as a master element. In other words, we have a well-defined function $\mathsf{master} : \mathcal{H} \to \{1, \ldots, P\}$ that is given by

$$\mathsf{master}(T) = p \quad \Leftrightarrow \quad T \in \mathsf{Ma}_{\ell(T)}(p) \ .$$

Let $T \in \mathsf{Ma}_k(p)$ be a parent master element. From the second result in lemma 4.1 and (A4) it follows that either all children are masters on the same processor p as T, or they are masters on some other processor q. In the latter case, the element T has a corresponding ghost element on processor q. Due to this property, in the parallel refinement algorithm below we use the strategy:

If a parent tetrahedron T has a ghost copy then operations that involve children of T are performed on the processor on which the ghost and the children are stored. (4.2)

From condition (A4) it follows that a child master element has its parent (as ghost or as master) on the same processor. Therefore we use the strategy:

Operations that involve the parent of T are performed on the processor on which the master element of T and its parent are stored. (4.3)

The first result in lemma 4.1 shows that every $T \in \mathcal{H}$ has at most one ghost copy. Moreover, due to (A5) all leaves ($T \in \mathcal{T}_J$) have no ghost copies. In this sense the overlap of tetrahedra between the processors is small.

REMARK 7. Comparing the distribution of the element data in UG ([6, 24]) with our approach, there are many similarities, but also some important differences. Of course in UG there are also ghost copies. For each UG master element its parent and neighbors (possibly as ghosts) are stored on the same processor. Related to this the ghosts are divided in two classes: vertical ghosts (overlap of parents) and horizontal ghosts (overlap of neighbors). In our approach we only have vertical ghosts, as we do not store copies of neighbors. Instead we have an overlapping storage of faces. Another significant difference is that in UG the children $\mathcal{K}(T)$ of a common parent T might be stored on different processors, whereas in our case the children $\mathcal{K}(T)$ are all on the same processor. Hence, in UG there might be up to $|\mathcal{K}(T)|$ ghost copies of T, whereas our format allows at most one.
The different data distribution formats have an impact on the actual algorithms. Hence, although the structure of our method is the same as the approach used in UG (V-cycle), there are significant differences in the parallel refinement methods. These are revealed only if one compares detailed descriptions of the algorithms (which are usually not presented in the literature). As one concrete example, we mention that in our algorithm there is no transfer of new horizontal ghosts like in UG (cf. [24]).

4.3. Parallel multilevel refinement algorithm. In figure 4.1 the parallel version of algorithm *SerRefinement* is given. For the subroutines that are new compared to the serial version of the refinement algorithm we use `a different font`.

The structure of the algorithm is the same as in the serial case. As for the serial algorithm, all tetrahedra T in the input have a corresponding $\mathsf{mark}(T) \in \{\mathsf{NoRef}, \mathsf{RegRef}, \mathsf{IrregRef}, \mathsf{Ref}, \mathsf{Del}\}$. In the output distributed hierarchical decomposition all tetrahedra have marks in the set $\{\mathsf{NoRef}, \mathsf{RegRef}, \mathsf{IrregRef}\}$. Below we discuss the subroutines used in the algorithm *ParRefinement*. Most of these subroutines are modifications of the subroutines (with the same name) that are used in the serial

```
Algorithm ParRefinement($\mathcal{G}_0(p), \dots, \mathcal{G}_J(p)$)
    for $k = J, \dots, 0$ do                                          // phase I
        DetermineMarks($\mathcal{G}_k(p)$);                                (1)
        CommunicateMarks($\mathcal{G}_k(p)$);                              (2)
        AccumulateEdgeCounter($\mathcal{G}_k(p)$);                         (3)
        MarksForClosure($\mathcal{G}_k(p)$);                               (4)

    for $k = 0, \dots, J$ do if $\mathcal{G}_k(p) \neq \emptyset$ then           // phase II
        if $k > 0$ then MarksForClosure($\mathcal{G}_k(p)$);              (5)
        if $k < J$ then Unrefine($\mathcal{G}_k(p)$);                      (6)
        Refine($\mathcal{G}_k(p)$);                                        (7)

    DetermineNewestLevel();                                           (8)
```

FIG. 4.1. *Parallel multilevel refinement algorithm.*

algorithm in section 3.3. For these we only explain the main differences between the parallel and the serial versions.

Let us first explain some notions that are used in the following to describe certain parallelization aspects: An object is said to be stored on a *processor boundary*, if it is stored on several processors at the same time. For these objects one often needs communication: The exchange of messages between corresponding local copies of objects on processor boundaries is called *interface communication*. The correlation of *local copies* and the corresponding *global objects* is realized by global identification numbers: Local copies (on different processors) with the same global ID represent the same global object.

The set of master elements on processor p which have ghost elements on some other processor is defined by

$$\mathsf{HasGh}_k(p) := \mathsf{Ma}_k(p) \cap \bigcup_{q=1,\dots,P} \mathsf{Gh}_k(q) \ .$$

This set plays an important role in the description of the subroutines for the parallel case. We note that if a tetrahedron T has a corresponding ghost element on some other processor then for certain operations which involve T, its children or its parent, one has to decide whether these are performed using T or the ghost copy of T. This decision is based on the two strategies (4.2) and (4.3) formulated in remark 6.

DetermineMarks. This subroutine is presented in figure 4.2 and is almost the same as for the serial case. Only $T \in \mathcal{G}_k(p) \setminus \mathsf{HasGh}_k(p)$ are involved in the block (4)–(10) since access to the children is needed (cf. (4.2)). In (5) and (8) edge counters are updated only by master tetrahedra to avoid multiple updates. The condition **if** *T is regular* can be checked without communication. For this it is important to note that *ghost elements are always regular*: if a ghost element T was irregular, it would not have any children, which contradicts (A5). For a master element its parent is stored on the same processor (cf. (A4)). Hence no communication is needed in this subroutine.

CommunicateMarks. This subroutine is given in figure 4.3. The marks from the ghost copies are communicated to the corresponding master tetrahedra (interface communication). Once this information is available, the marks and edge counters of the tetrahedra $T \in \mathsf{HasGh}_k(p)$ which where not treated in *DetermineMarks* are

```
Function DetermineMarks($\mathcal{G}_k(p)$)
    for $T \in \mathcal{G}_k(p)$ do
        if status($T$) = NoRef then
            if $T$ is regular and mark($T$) = Ref then
                mark($T$) := RegRef;                                        (1)
                increase edge counters;                                     (2)
            if $k = 0$ and mark($T$) = Del then
                mark($T$) := NoRef;                                         (3)
        else if $T \notin$ HasGh$_k(p)$ then
            if status($T$) = RegRef then
                if $\forall T' \in \mathcal{K}(T)$ : mark($T'$) = Del then
                    mark($T$) := NoRef;                                     (4)
                    if $T \notin$ Gh$_k(p)$ then decrease edge counters;    (5)
                for $T' \in \mathcal{K}(T)$ do
                    if mark($T'$) = Del then
                        mark($T'$) := NoRef;                                (6)
            else // status($T$) = IrregRef
                if $\exists T' \in \mathcal{K}(T)$ : mark($T'$) = Ref
                or an edge of a child, which is not an edge of $T$,
                    is marked for refinement then
                    mark($T$) := RegRef;                                    (7)
                    if $T \notin$ Gh$_k(p)$ then increase edge counters;    (8)
                else
                    mark($T$) := NoRef;                                     (9)
                for $T' \in \mathcal{K}(T)$ do
                    mark($T'$) := NoRef;                                    (10)
```

FIG. 4.2. *Subroutine DetermineMarks; parallel case.*

modified, if necessary. Note that, due to (A5), for $T \in \mathsf{HasGh}_k(p)$ always $\mathsf{status}(T) \neq$ NoRef holds.

```
Function CommunicateMarks($\mathcal{G}_k(p)$)
    for $T \in$ Gh$_k(p)$ do
        $m_T$ := mark($T$);                                    (1)
    for ( $q = 1, \dots, P, \quad q \neq p$ ) do
        send $\{m_T : T \in$ Gh$_k(p) \cap$ Ma$_k(q)\}$
        to processor $q$;                                      (2)
    Receive corresponding messages
    from other processors;                                     (3)
    for $T \in$ HasGh$_k(p)$ do
        mark($T$) := $m_T$;                                    (4)
        if status($T$) = RegRef then
            if $m_T \neq$ RegRef then
                decrease edge counters;                        (5)
        else // status($T$) = IrregRef
            if $m_T$ = RegRef then
                increase edge counters;                        (6)
```

FIG. 4.3. *Subroutine* `CommunicateMarks`.

AccumulateEdgecounter. For an edge E that is stored on several processors the value of its edge counter $C(E)$ is stored in a distributed manner, i.e. each local copy stores a contribution of the global value. Thus in order to obtain the global counter value $C(E)$ on each local copy of E, the local values have to be added up among the local copies. For this purpose interface communication involving the edges is needed.

MarksForClosure. This subroutine is the same as the one in the serial case in figure 3.8 (one only has to replace $\mathcal{G}_k$ by $\mathcal{G}_k(p)$). Since after the call of `AccumulateEdge-counter`$(\mathcal{G}_k(p))$ the edge refinement pattern is available on each tetrahedron $T \in \mathcal{G}_k(p)$ the marks can be set without communication.

Unrefine. This subroutine is shown in figure 4.4. In the first for-loop $T \in \mathsf{HasGh}_k(p)$ are skipped since in (2) access to the children is needed (see (4.2)). Ghost elements that are marked for no refinement are deleted in (3) in order to ensure (A5). The second for-loop (4) is needed because it may happen that for $T' \in \mathsf{Gh}_{k+1}(p)$ there is no parent on the same processor that removed the deletion labels from T' in (2). Note that the deletion of local objects that have copies on other processors causes communication since all copies have to be informed about the local deletion.

```
Function Unrefine($\mathcal{G}_k(p)$)
    Label all tetrahedra, vertices, edges and faces on level $k+1$ for deletion;      (1)

    for $T \in \mathcal{G}_k(p) \setminus \mathsf{HasGh}_k(p)$ do if $\mathsf{status}(T) \neq \mathsf{NoRef}$ then
        if $\mathsf{mark}(T) = \mathsf{status}(T)$ then
            Remove all deletion labels of the
            children of $T$ and of their vertices, edges, faces;                     (2)
        else if $T \in \mathsf{Gh}_k(p)$ and $\mathsf{mark}(T) = \mathsf{NoRef}$ then
            Remove $T$ and its vertices, edges, faces,
            that are not needed anymore by other tetrahedra;                          (3)

    for $T \in \mathsf{Gh}_{k+1}(p)$ do
        Remove all deletion labels of $T$ and of its vertices, edges, faces;          (4)

    Remove all tetrahedra, vertices, edges and faces on level $k+1$,
    that are labeled for deletion;                                                    (5)
```

FIG. 4.4. *Subroutine Unrefine; parallel case.*

Refine. This subroutine is given in figure 4.5. It is almost the same as for the serial case except that the for-loop only iterates over $\mathcal{G}_k(p) \setminus \mathsf{HasGh}_k(p)$, cf. (4.2). We emphasize that local objects created on processor boundaries have to be identified with each other. Consider, for example, an unrefined edge E which is stored on two different processors, say p and q, with a counter $C(E)$ that has changed and is now positive. Then for both local copies of E new midvertices V_p and V_q are created independently on processor p and q, respectively. The information that V_p and V_q represent the same global object has to be communicated between the processors p and q by identifying both midvertices with each other. The identification of edges and faces can be done in a similar way. In our implementation we use the DDD library to take care of this identification (cf. remark 10).

DetermineNewestLevel. In this subroutine, which is described in figure 4.6, the new global finest level index $\in \{J-1, J, J+1\}$ is determined. For this an `MPI` $-$ `Allreduce` operation is used in (4).

REMARK 8. Note that all communication in *ParRefinement* is done per level, so the number of communication steps is proportional to J. The *amount* of communication depends on the number of overlapping objects, i.e. the number of ghost copies

```
Function Refine($\mathcal{G}_k(p)$)
    if $k = J$ then $\mathcal{G}_{k+1}(p) := \emptyset$;                                    (1)

    for $T \in \mathcal{G}_k(p) \setminus \mathsf{HasGh}_k(p)$ do
        if $\mathsf{mark}(T) \neq \mathsf{status}(T)$ then
            Refine $T$ according to $\mathsf{mark}(T)$, i.e.
            create children of $T$ as master elements;                      (2)
            $\mathsf{status}(T) := \mathsf{mark}(T)$;                                          (3)
            for $T' \in \mathcal{K}(T)$ do
                Find existing vertices, edges and faces of $T'$;            (4)
                create missing vertices, edges and faces of $T'$;           (5)
                $\mathsf{status}(T') := \mathsf{NoRef}$;                                      (6)
```

FIG. 4.5. *Subroutine Refine; parallel case.*

```
Function DetermineNewestLevel()
    if $\mathcal{G}_J(p) = \emptyset$ then $J_p := J - 1$;                   (1)
    else if $\mathcal{G}_{J+1}(p) \neq \emptyset$ then $J_p := J + 1$;       (2)
    else $J_p := J$;                                     (3)

    Determine $J' := \max_{1 \le p \le P} J_p$ ;                  (4)

    if $J_p < J'$
        for $k = J_p + 1, \ldots, J'$ do
            $\mathcal{G}_k(p) := \emptyset$;                            (5)
```

FIG. 4.6. *Subroutine* `DetermineNewestLevel`.

and the size of the processor boundaries.

REMARK 9. We note that *ghost copies are not created within the parallel refinement algorithm.* These are generated only in a data migration procedure that is used for load balancing. It is clear that for efficient parallel (local) multilevel refinement the issue of dynamic load balancing is of major importance (cf. [8, 24]). In this paper we do not discuss the load balancing and data migration algorithm that we use. This will be presented in a separate paper. The main idea is as follows. The load balancing is performed by means of a graph partitioning method which assumes the dual graph G of a triangulation as input. In order to ensure that the children of a common parent are all assigned to the same processor, they are combined to one multi-node, inducing a reduced dual graph G' which is then passed to the graph partitioner. Based on the output of the graph partitioning method certain tetrahedra are migrated and certain ghost copies are created. *It can be shown that given an input admissible hierarchical decomposition the data migration algorithm is such that its output is again an admissible hierarchical decomposition.*

REMARK 10. We briefly comment on some implementation issues. The management of the distributed data is handled by the library DDD (which stands for *dynamic distributed data*), which has been developed by BIRKEN [14, 15]. This library is also used in the package UG [7] for the same task. The behaviour of DDD can be adjusted to the user's requirements by means of several user-defined handler functions. The functionality of DDD includes the deletion of objects and the transfer of local objects from one processor to another (transfer module), interface communication across processor boundaries (interface module) and the identification of local objects from different processors (identification module). All communication is bundled such that the number of exchanged messages is kept small. All parallel communication and administration actions occuring in *ParRefinement* (e.g., the accumulation of edge counters by interface communication, the consistent deletion of local objects or the identification of local objects with each other) can be handled by DDD.

5. Analysis of the parallel refinement algorithm. In this section we analyze the algorithm *ParRefinement* applied to an input *admissible* hierarchical decomposition

$$\tilde{\mathcal{H}}^{\mathrm{old}} = (\mathcal{H}^{\mathrm{old}}(1), \dots, \mathcal{H}^{\mathrm{old}}(P)) \tag{5.1}$$

as in (4.1) with $\mathcal{H}^{\mathrm{old}}(p) = (\mathcal{G}_0^{\mathrm{old}}(p), \dots, \mathcal{G}_J^{\mathrm{old}}(p))$. The main result we prove (theorem 5.7) is that the output of this algorithm

$$\tilde{\mathcal{H}}^{\mathrm{new}} = (\mathcal{H}^{\mathrm{new}}(1), \dots, \mathcal{H}^{\mathrm{new}}(P)) \tag{5.2}$$

is again an admissible hierarchical decomposition.

First we prove that the algorithm *ParRefinement* is well-defined:

THEOREM 5.1. *The algorithm* ParRefinement *applied to an admissible hierarchical decomposition is well-defined.*

Proof. We have to show that all data that are needed in the subroutines of *ParRefinement* applied to $\mathcal{G}_k(p)$ are available on processor p. By definition a tetrahedron $T \in \mathcal{G}_k(p)$ is stored (as master or ghost) on processor p. A tetrahedron is always stored together with its vertices, edges and faces. Hence information on these subsimplices is available on processor p if the tetrahedron is available on processor p.

For $T \in \mathcal{G}_k(p)$ the two critical points are references to its parent and references to its children. The former is needed only to check the condition **if** T *is regular*, which occurs at several places.

We first consider this condition. If T is a ghost element then it must be regular (hence the condition must be fulfilled), since if we assume T to be irregular then T has no children and a contradiction follows from (A5). If T is a master element, then due to (A4) its parent is stored (as ghost or master) on processor p. Hence the reference to its parent is available on processor p and the condition can be checked.

We now consider the references to the children of T that occur at several places in the subroutines. A look at the subroutines shows that such a reference is needed only for $T \in \mathcal{G}_k(p) \setminus \mathsf{HasGh}_k(p)$. Such a T is either a master element (on processor p) which does not have a ghost copy on another processor or a ghost element. In the former case it follows using (A4) that children of T must be master elements on processor p. If T is a ghost element then it follows from (A6) that the children are stored on processor p. We conclude that in all cases the references to children are available on processor p. $\square$

It is straightforward to check that for a given hierarchical decomposition of an input multilevel triangulation the serial algorithm *SerRefinement* and its parallel version *ParRefinement* yield the same output triangulation. In [11, 12] it is proved that this output is again a multilevel triangulation. Hence the distributed hierarchical decomposition in (5.2), which corresponds to the output multilevel triangulation is well-defined. In the remainder of this section we will prove that this output distributed hierarchical decomposition is admissible. We first show that the condition (A1) is satisfied. Here and in the remainder we always assume that *the input distributed hierarchical decomposition* (5.1) *is admissible.*

LEMMA 5.2. *The output distributed hierarchical decomposition (5.2) of algorithm ParRefinement satisfies condition (A1).*

Proof. The new set $\mathcal{G}_k^{\text{new}}(p)$ is obtained from the old one $\mathcal{G}_k^{\text{old}}(p)$ by removing elements (in *Unrefine*) and by creating new *master* elements (in *Refine*). The removal of elements does not destroy property (A1). If a new tetrahedron T is added to $\mathcal{G}_k^{\text{old}}(p)$ we obtain $\mathcal{G}_k'(p) := \mathcal{G}_k(p) \cup \{T\}$ and the partitioning

$$\mathsf{Ma}_k'(p) := \mathsf{Ma}_k(p) \cup \{T\}\ , \qquad \mathsf{Gh}_k'(p) := \mathsf{Gh}_k(p)\ ,$$

which still has property (A1). ☐

In order to prove that the other conditions (A2)-(A6) also hold we need a few lemmas. We use the notation $F(T)$ for the parent of T.

LEMMA 5.3. *Let $k \geq 1$ and let $T \in \mathcal{G}_k^{\text{new}}$ be a tetrahedron such that $T \notin \mathcal{G}_k^{\text{old}}$. Then the following holds:*

$$\exists\, p: \quad T \in \mathsf{Ma}_k^{\text{new}}(p) \tag{5.3}$$

$$T \in \mathsf{Ma}_k^{\text{new}}(p) \quad \Rightarrow \quad F(T) \in \mathcal{G}_{k-1}^{\text{new}}(p) \tag{5.4}$$

Proof. A new tetrahedron $T \in \mathcal{G}_k^{\text{new}} \setminus \mathcal{G}_k^{\text{old}}$ can only be constructed in the subroutine *Refine*. There it is created as a master element on some processor p. Hence, the result (5.3) holds.

Let $T \in \mathsf{Ma}_k^{\text{new}}(p)$ be a new master element which has a parent, i.e., it is obtained by applying a refinement rule to its parent $F(T) \in \mathcal{G}_{k-1}^{\text{old}}$. Since T is on processor p, this refinement must be performed in the call of the subroutine *Refine*$(\mathcal{G}_{k-1}^{\text{old}}(p))$, and thus $F(T) \in \mathcal{G}_{k-1}^{\text{old}}(p)$. The element $F(T)$ can only be removed in the subroutine *Unrefine* on level $\ell < k$. This can not happen because the second phase of the refinement algorithm ((5)-(7)) is a bottom-up procedure. We conclude that $F(T) \in \mathcal{G}_{k-1}^{\text{new}}(p)$ holds. ☐

LEMMA 5.4. Let $k \geq 0$. For $T \in \mathcal{G}_k^{\text{new}} \cap \mathcal{G}_k^{\text{old}}$ the following holds:

$$T \in \mathsf{Ma}_k^{\text{old}}(p) \quad \Leftrightarrow \quad T \in \mathsf{Ma}_k^{\text{new}}(p) \tag{5.5}$$

$$T \in \mathcal{G}_k^{\text{old}}(p) \ \text{ and } \ \mathcal{K}^{\text{new}}(T) \neq \emptyset \quad \Rightarrow \quad T \in \mathcal{G}_k^{\text{new}}(p) \tag{5.6}$$

Proof. Consider a tetrahedron $T \in \mathcal{G}_k^{\text{old}} \cap \mathcal{G}_k^{\text{new}}$. The result in (5.5) follows from the fact that nowhere in the algorithm *ParRefinement* a tetrahedron is moved from one processor to another or a master element is changed to a ghost element.

We now consider (5.6). If $T \in \mathsf{Ma}_k^{\text{old}}(p)$ then the result directly follows from (5.5). Hence, we only have to consider a ghost element $T \in \mathsf{Gh}_k^{\text{old}}(p)$ with $\mathcal{K}^{\text{new}}(T) \neq \emptyset$. Because the tetrahedron T has children in $\tilde{\mathcal{H}}^{\text{new}}$ we have $\mathsf{mark}^{\text{new}}(T) \neq \mathsf{NoRef}$. Assume that $T \notin \mathcal{G}_k^{\text{new}}(p)$ holds. Since tetrahedra are not moved between processors it follows that T has been removed from $\mathcal{G}_k^{\text{old}}(p)$. This can be caused only by the assignments (5) in *Unrefine*$(\mathcal{G}_{k-1}^{\text{old}}(p))$ or (3) in *Unrefine*$(\mathcal{G}_k^{\text{old}}(p))$. We first consider the latter case. From the bottom-up structure of the second phase of the refinement algorithm it follows that $\mathsf{mark}(T)$ is not changed after the call of *Unrefine*$(\mathcal{G}_k^{\text{old}}(p))$. Thus, in this call we have $\mathsf{mark}(T) \neq \mathsf{NoRef}$ and the assignment (3) is not applied.

We now treat the other case. As outlined above we only have to consider a ghost element $T \in \mathsf{Gh}_k^{\text{old}}(p)$. Thus assignment (4) in *Unrefine*$(\mathcal{G}_{k-1}^{\text{old}}(p))$ is applied to T and so T won't be removed in assignment (5) of this routine. Since there are no other cases left, we conclude that T is not removed from $\mathcal{G}_k^{\text{old}}(p)$, which yields a contradiction. ☐

The result in (5.5) shows that master elements from $T \in \mathcal{G}_k^{\text{new}} \cap \mathcal{G}_k^{\text{old}}$ remain on the same processor. The result in (5.6) yields that elements from $T \in \mathcal{G}_k^{\text{new}} \cap \mathcal{G}_k^{\text{old}}$

that are refined in $\tilde{\mathcal{H}}^{\text{new}}$ remain on the same processor. The refinement condition $\mathcal{K}^{\text{new}}(T) \neq \emptyset$ is necessary (there may be ghost elements in $\mathcal{G}_k^{\text{old}}$ that are removed).

The next lemma shows that ghost elements in $\mathcal{G}_k^{\text{new}}$ on processor p must be ghost elements in $\mathcal{G}_k^{\text{old}}$ on the same processor.

Lemma 5.5. The following holds:

$$T \in \mathsf{Gh}_k^{\text{new}}(p) \quad \Rightarrow \quad T \in \mathsf{Gh}_k^{\text{old}}(p)$$

Proof. This can be concluded from the following two properties. Firstly, tetrahedra are not moved from one processor to another. Secondly, new tetrahedra (created in the subroutine *Refine*) always are masters. $\square$

We now formulate a main result of our analysis.

Theorem 5.6. *Let the input hierarchical decomposition $\tilde{\mathcal{H}}^{\text{old}}$ be admissible, i.e., it satisfies the conditions (A1)-(A6) for the levels $k = 0, \ldots, J$. Then for the output hierarchical decomposition $\tilde{\mathcal{H}}^{\text{new}}$ the conditions (A2) -(A6) are satisfied for the levels $k = 0, \ldots, J-1$.*

Proof. Note that the levels $k = 0, \ldots, J-1$ exist in $\tilde{\mathcal{H}}^{\text{new}}$. Since the distributed input hierarchical decomposition $\tilde{\mathcal{H}}^{\text{old}}$ is admissible we can use that on the levels $k = 0, \ldots, J$ the conditions (A1)-(A6) hold for $\tilde{\mathcal{H}}^{\text{old}}$. These properties are denoted by (A1)$^{\text{old}}$-(A6)$^{\text{old}}$.

(A2). By definition we have $\mathcal{G}_k^{\text{new}} = \bigcup_{p=1}^{P} \mathcal{G}_k^{\text{new}}(p)$ and $\mathsf{Ma}_k^{\text{new}}(p) \subset \mathcal{G}_k^{\text{new}}(p)$. From this we get $\bigcup_{p=1}^{P} \mathsf{Ma}_k^{\text{new}}(p) \subseteq \mathcal{G}_k^{\text{new}}$. For the other inclusion $\supseteq$ we consider $T \in \mathcal{G}_k^{\text{new}}$. If $T \in \mathcal{G}_k^{\text{old}}$ then using (A2)$^{\text{old}}$ we have:

$$\exists\, p \quad T \in \mathsf{Ma}_k^{\text{old}}(p)$$

and thus, from lemma 5.4 it follows that $T \in \mathsf{Ma}_k^{\text{new}}(p)$ holds. Now consider the case $T \notin \mathcal{G}_k^{\text{old}}$. On level 0 no new tetrahedra are created and thus $k \geq 1$ must hold. From lemma 5.3 we obtain:

$$\exists\, p \quad T \in \mathsf{Ma}_k^{\text{new}}(p) \ .$$

We conclude that the inclusion $\bigcup_{p=1}^{P} \mathsf{Ma}_k^{\text{new}}(p) \supseteq \mathcal{G}_k^{\text{new}}$ holds, too.

(A3). We have to prove:

$$\forall\, p, q: \qquad \mathsf{Ma}_k^{\text{new}}(p) \cap \mathsf{Ma}_k^{\text{new}}(q) \neq \emptyset \ \Rightarrow \ p = q \tag{5.7}$$

This will be proved using an induction argument. For $k = 0$ we have $\mathsf{Ma}_0^{\text{new}}(r) = \mathsf{Ma}_0^{\text{old}}(r)$ for $r = 1, \ldots, P$. Thus the result (5.7) for $k = 0$ follows from (A3)$^{\text{old}}$.

Now we take $k \geq 1$ and assume that the result in (5.7) holds for the lower levels ℓ, $0 \leq \ell < k$. Consider $T \in \mathsf{Ma}_k^{\text{new}}(p) \cap \mathsf{Ma}_k^{\text{new}}(q)$.

If $T \in \mathcal{G}_k^{\text{old}}$, then from lemma 5.4 we obtain

$$T \in \mathsf{Ma}_k^{\text{old}}(p) \quad \text{and} \quad T \in \mathsf{Ma}_k^{\text{old}}(q) \ .$$

From (A3)$^{\text{old}}$ it follows that $p = q$ holds.

We now treat the other, more involved case $T \notin \mathcal{G}_k^{\text{old}}$. Then T must have a parent $F(T)$. From lemma 5.3 we get

$$F(T) \in \mathcal{G}_{k-1}^{\text{new}}(p) \cap \mathcal{G}_{k-1}^{\text{new}}(q) \ .$$

This parent can be stored as master or as ghost. We analyze the three possible situations:

1. Both on procesor p and processor q the parent $F(T)$ is stored as a master element: $F(T) \in \mathsf{Ma}_{k-1}^{\mathrm{new}}(p) \cap \mathsf{Ma}_{k-1}^{\mathrm{new}}(q)$. From (5.7) on level $k-1$ it follows that $p = q$ holds.
2. Both on procesor p and processor q the parent $F(T)$ is stored as a ghost element: $F(T) \in \mathsf{Gh}_{k-1}^{\mathrm{new}}(p) \cap \mathsf{Gh}_{k-1}^{\mathrm{new}}(q)$. From lemma 5.5 it follows that

   $$F(T) \in \mathsf{Gh}_{k-1}^{\mathrm{old}}(p) \cap \mathsf{Gh}_{k-1}^{\mathrm{old}}(q) \ .$$

   From the first result in lemma 4.1 we conclude that $p = q$ holds.
3. Assume that $F(T)$ is stored as ghost on one processor and as a master on another one:

   $$F(T) \in \mathsf{Gh}_{k-1}^{\mathrm{new}}(p) \quad \text{and} \quad F(T) \in \mathsf{Ma}_{k-1}^{\mathrm{new}}(q) \ , \quad p \neq q.$$

   From lemma 5.5 we get $F(T) \in \mathsf{Gh}_{k-1}^{\mathrm{old}}(p)$ and thus also $F(T) \in \mathcal{G}_{k-1}^{\mathrm{old}}$. Application of lemma 5.4 yields $F(T) \in \mathsf{Ma}_{k-1}^{\mathrm{old}}(q)$ and we conclude:

   $$F(T) \in \mathsf{HasGh}_{k-1}^{\mathrm{old}}(q) \ . \tag{5.8}$$

   The new tetrahedron $T \in \mathsf{Ma}_{k}^{\mathrm{new}}(q)$ can only be created as refinement of $F(T)$ in instruction (2) in the call of the subroutine *Refine*$(\mathcal{G}_{k-1}^{\mathrm{old}}(q))$. However, this instruction is only performed if $F(T) \notin \mathsf{HasGh}_{k-1}^{\mathrm{old}}(q)$. Thus we obtain a contradiction with (5.8). We conclude that this third case can not occur.

(A4). Take $T \in \mathcal{G}_k^{\mathrm{new}}$. If $\mathcal{K}^{\mathrm{new}}(T) = \emptyset$ there is nothing to prove. We consider the situation $\mathcal{K}^{\mathrm{new}}(T) \neq \emptyset$, $T' \in \mathcal{K}^{\mathrm{new}}(T)$ with $T' \in \mathsf{Ma}_{k+1}^{\mathrm{new}}(p)$.

We first treat the case $T' \in \mathcal{G}_{k+1}^{\mathrm{old}}$. Then for its parent T we have $T \in \mathcal{G}_k^{\mathrm{old}}$ and $T' \in \mathcal{K}^{\mathrm{old}}(T)$. Take q such that $T' \in \mathsf{Ma}_{k+1}^{\mathrm{old}}(q)$. Using lemma 5.4 we get $T' \in \mathsf{Ma}_{k+1}^{\mathrm{new}}(q)$. From the uniqueness property (A3) we conclude $q = p$. We thus have $T' \in \mathsf{Ma}_{k+1}^{\mathrm{old}}(p)$, and together with (A4)$^{\mathrm{old}}$ this yields $T \in \mathcal{G}_k^{\mathrm{old}}(p)$. Now note that $\mathcal{K}^{\mathrm{new}}(T) \neq \emptyset$ and use the second result in lemma 5.4. This implies

$$T \in \mathcal{G}_k^{\mathrm{new}}(p) \ .$$

We now consider the case $T' \notin \mathcal{G}_{k+1}^{\mathrm{old}}$. From lemma 5.3 we obtain:

$$\exists\, q \quad T' \in \mathsf{Ma}_{k+1}^{\mathrm{new}}(q) \ .$$

Due to the uniqueness property (A3) we have $q = p$. Application of the second result in lemma 5.3 yields

$$T = F(T') \in \mathcal{G}_k^{\mathrm{new}}(p) \ .$$

So in both cases we have that the parent T is stored on the same processor p as T'.

(A5). Take $T \in \mathsf{Gh}_k^{\mathrm{new}}(p)$. Lemma 5.5 implies

$$T \in \mathsf{Gh}_k^{\mathrm{old}}(p) \ ,$$

and $\mathcal{K}^{\mathrm{old}}(T) \neq \emptyset$ because of (A5)$^{\mathrm{old}}$. Assume that $\mathcal{K}^{\mathrm{new}}(T) = \emptyset$ holds. This implies $\mathsf{mark}^{\mathrm{new}}(T) = \mathsf{NoRef}$. In the second phase of the refinement algorithm the mark of T is not changed after the call of *Unrefine*$(\mathcal{G}_k(p))$, and thus $\mathsf{mark}(T) = \mathsf{NoRef}$ during this call. In the instruction (2) of this call the element T is then removed, which implies $T \notin \mathsf{Gh}_k^{\mathrm{new}}(p)$. We obtain a contradiction and conclude that $\mathcal{K}^{\mathrm{new}}(T) \neq \emptyset$ holds.

(A6). Take $T \in \mathsf{Gh}_k^{\mathrm{new}}(p)$. Lemma 5.5 implies $T \in \mathsf{Gh}_k^{\mathrm{old}}(p)$. From $(\mathrm{A5})^{\mathrm{old}}$ it follows that $\mathcal{K}^{\mathrm{old}}(T) \neq \emptyset$. From (A5) we have $\mathcal{K}^{\mathrm{new}}(T) \neq \emptyset$. Let

$$T' \in \mathcal{K}^{\mathrm{new}}(T) \subset \mathcal{G}_{k+1}^{\mathrm{new}}$$

be one of the children of T in $\tilde{\mathcal{H}}^{\mathrm{new}}$.

First consider the case $T' \in \mathcal{K}^{\mathrm{old}}(T)$. Due to $T \in \mathsf{Gh}_k^{\mathrm{old}}(p)$ and $(\mathrm{A6})^{\mathrm{old}}$ we get $T' \in \mathsf{Ma}_{k+1}^{\mathrm{old}}(p)$ and lemma 5.4 yields $T' \in \mathsf{Ma}_{k+1}^{\mathrm{new}}(p)$.

We now treat the other case, $T' \notin \mathcal{K}^{\mathrm{old}}(T)$. Since $F(T') = T \in \mathsf{Gh}_k^{\mathrm{old}}(p)$ the new child tetrahedron T' must have been created in *Refine*$(\mathcal{G}_k(p))$, instruction (2). Thus $T' \in \mathsf{Ma}_{k+1}^{\mathrm{new}}(p)$ holds.

We conclude that $\mathcal{K}^{\mathrm{new}}(T) \subset \mathsf{Ma}_{k+1}^{\mathrm{new}}(p)$. ☐

Finally, we prove the main result:

THEOREM 5.7. *The output distributed hierarchical decomposition (5.2) of algorithm ParRefinement satisfies the conditions (A1)-(A6).*

Proof. The result concerning (A1) follows from lemma 5.3. It remains to prove that the conditions (A2) – (A6) are satisfied for the sets $\mathcal{G}_k^{\mathrm{new}}(p)$, $k = 0, \dots, J^{\mathrm{new}}$. Note that $J^{\mathrm{new}} \in \{J-1, J, J+1\}$. We consider these three cases.

$\mathbf{J}^{\mathrm{new}} = \mathbf{J} - \mathbf{1}$. For this situation the desired result is proved in theorem 5.6.

$\mathbf{J}^{\mathrm{new}} = \mathbf{J}$. From theorem 5.6 it follows that the sets $\mathcal{G}_k^{\mathrm{new}}(p), k = 0, \dots, J-1$ satisfy the conditions (A2) -(A6), hence we only have to consider $\mathcal{G}_J^{\mathrm{new}}(p)$. First note that, due to $(\mathrm{A5})^{\mathrm{old}}$ we have $\mathcal{G}_J^{\mathrm{old}}(p) = \mathsf{Ma}_J^{\mathrm{old}}(p)$, i.e., the input finest level contains only master elements. Now consider an element $T \in \mathcal{G}_J^{\mathrm{new}}(p)$. Then either $T \in \mathcal{G}_J^{\mathrm{old}}(p)$, in which case T is a master element or T is a child of some element $F(T) \in \mathcal{G}_{J-1}^{\mathrm{new}}$. In the latter case T is created in *Refine* (2) and must be a master element. We conclude that $\mathcal{G}_J^{\mathrm{new}}(p) = \mathsf{Ma}_J^{\mathrm{new}}(p)$ and $\mathsf{Gh}_J^{\mathrm{new}}(p) = \emptyset$, i.e., there are no ghost elements in the output on level J. One easily verifies that for this situation the conditions (A1)-(A6) are satisfied for level J (as for $\mathcal{G}_0$ in remark 5).

$\mathbf{J}^{\mathrm{new}} = \mathbf{J} + \mathbf{1}$. The arguments used in the case $J^{\mathrm{new}} = J$ still hold and thus it follows that the conditions (A1)-(A6) hold for the levels $k = 0, \dots, J$. We only have to consider the set $\mathcal{G}_{J+1}^{\mathrm{new}}(p)$. This set contains only new tetrahedra, i.e. elements which are created in *Refine* (2) by refinement of elements from $\mathcal{G}_J^{\mathrm{new}}$. Thus all these elements are masters: $\mathcal{G}_{J+1}^{\mathrm{new}}(p) = \mathsf{Ma}_{J+1}^{\mathrm{new}}(p)$ and $\mathsf{Gh}_{J+1}^{\mathrm{new}}(p) = \emptyset$. The same argument as used in the case $J^{\mathrm{new}} = J$ yields that the conditions (A1)-(A6) are satisfied for $\mathcal{G}_{J+1}^{\mathrm{new}}(p)$. ☐

6. Numerical experiments.

In this section we present results of a few numerical experiments. The computations were executed on the RWTH SunFire SMP cluster, using up to 64 900-MHz-processors. 1 GB RAM per processor was reserved by the batch system when starting a parallel job. The SMP nodes are connected to each other by Gigabit Ethernet, messages between processors of the same node are exchanged through the shared memory. Because of the multi-user mode one can not expect optimal runtimes. All runs were performed twice, then for each number of processors P the average of the measured runtimes was taken. The parallel runtimes are compared to the serial runtime in terms of the following definition to obtain a rough measure for the scalability of the parallel algorithm.

DEFINITION 9 (Scaled efficiency). Consider an algorithm of complexity $\mathcal{O}(n)$ where n denotes the characteristic problem size. For a given number of processor P let $T_n(P)$ denote the runtime of the parallel algorithm on P processors for a problem of total size n. Choose a *fixed local problem size* n' (based on the memory available per processor). The ratio

$$E^{sc}_{n'}(P) := \frac{T_{n'}(1)}{T_{P\cdot n'}(P)}$$

of serial runtime to parallel runtime on P processors for a fixed local problem size n' is called the *scaled efficiency*.

Note that the refinement algorithm is of complexity $\mathcal{O}(n)$ where n is the number of tetrahedra.

6.1. Experiment 1: global refinement. First, we consider the global refinement of the unit cube $\Omega = [0,1]^3$. For P processors the initial triangulation $\mathcal{T}_0$ consists of $P\cdot 384$ tetrahedra. Thus, the problem size grows proportional to the processor number and we have a fixed local problem size. This triangulation is refined globally four times. The resulting finest triangulation $\mathcal{T}_4$ then consists of $P \cdot 1\,572\,864$ tetrahedra, the whole final multilevel triangulation $\mathcal{M}$ contains $P \cdot 1\,797\,504$ tetrahedra.

The parallel runtimes (in seconds) for the last refinement step and the scaled efficiency for several numbers of processors P are shown in table 6.1. Satisfactory efficiencies are obtained and thus the parallel refinement algorithm turns out to be scalable for uniform refinement (at least, up to $P = 64$ processors).

P	1	2	4	8	16	32	64
$T(P)$	69.45	70.96	71.31	73.42	79.48	86.48	94.79
$E^{sc}(P)$	100%	98%	97%	95%	87%	84%	80%

TABLE 6.1
Experiment 1: global refinement, last refinement step.

6.2. Experiment 2: local refinement. In the second and third experiment we treat a problem with local refinement. In the second experiment we do not use load balancing, whereas in the third experiment a load distribution is used. We again consider the unit cube $\Omega = [0,1]^3$ with the same initial triangulation as in experiment 1. Let $B \subset \Omega$ be the ball with radius $r = 0.3$ centered at $(0.4, 0.4, 0.4)$. Before applying the refinement algorithm only those tetrahedra are marked for refinement, whose barycenter is located within B. This is repeated four times. The parallel runtimes (in seconds) for the last refinement step are presented in table 6.2. After the last refinement the number of tetrahedra per processor ranges from 384 up to 927 832. Due to this heavy load imbalance between the processors two effects can be observed. On the one hand efficiency is very poor even for fairly low processor numbers due to the fact that for some processors the local grid is refined almost uniformly whereas on other processors only few tetrahedra are refined. Thus some of the processors are idle waiting for others with much more computational load. On the other hand a large amount of the available memory will not be used, so the theoretical maximum problem size $n_{\max}$, which is determined by the overall memory supply of the parallel machine, cannot be reached without a suitable load balancing strategy.

6.3. Experiment 3: local refinement with load balancing. Because of the disadvantages mentioned above it is clear, that one has to use load balancing to obtain better efficiency and better memory utilization. In the third experiment we consider the situation as in experiment 2 but now additionally apply load balancing before

P	1	2	4	8	16	32	64
$T(P)$	7.44	11.11	17.03	29.42	42.52	57.06	112.23
$E^{sc}(P)$	100%	67%	44%	25%	17%	13%	7%

TABLE 6.2

Experiment 2: local refinement without load balancing, last refinement step.

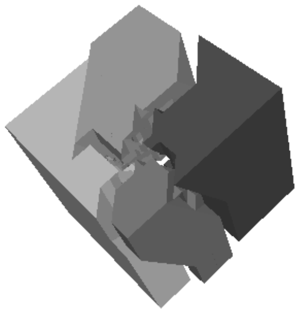


FIG. 6.1. *Distributed triangulation.*

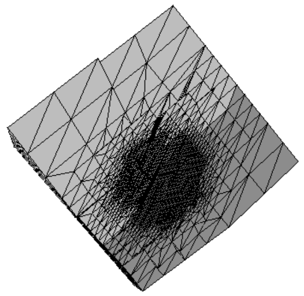


FIG. 6.2. *Interior.*

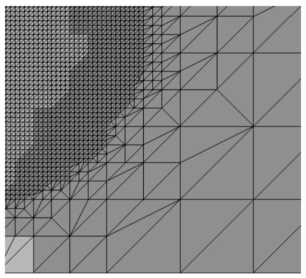


FIG. 6.3. *Detail.*

the last refinement step. In contrast to the previous two experiments ghost copies appear which are created during the load balancing phase. The parallel runtimes (in seconds) for the last refinement step are shown in table 6.3. One of course obtains much better results as for the imbalanced case in table 6.2. Nevertheless we get worse efficiency with local refinement as compared to the results for uniform refinement in table 6.1. This is not surprising as there is more communication (because of the ghost copies) and at the same time much less computational work: The average number of tetrahedra per processor after the last refinement step is about 200 000 whereas in the first experiment this number is about $1.8 \cdot 10^6$.

P	1	2	4	8	16	32	64
$T(P)$	8.07	8.63	9.37	10.47	12.14	14.73	23.97
$E^{sc}(P)$	100%	94%	86%	77%	67%	55%	34%

TABLE 6.3

Experiment 3: local refinement with load balancing, last refinement step.

Figures 6.1–6.3 show the distributed triangulation on $P = 4$ processors after the last refinement.

REMARK 11. In an adaptive parallel CFD code the parallel multilevel refinement algorithm is used in combination with discretization routines (creation of the stiffness matrix) and solution routines (iterative solvers). Usually the bulk of the arithmetic work is needed in the solution routine. In such a setting a deterioration of the parallel efficiency as in table 6.3 will probably hardly influence the overall parallel efficiency. The latter is mainly determined by the parallel efficiency of the solution method. Experiments with the UG-package have shown that even for very complex problems a parallel multilevel grid refinement method in combination with dynamic load balancing and fast iterative solvers can have a high overall parallel efficiency (cf. [9, 10, 24]).

REFERENCES

[1] E. Bänsch: *Local mesh refinement in 2 and 3 dimensions*, Impact of Computing in Science and Engineering 3, 1991, 181–191.

[2] R.E. Bank: *PLTMG: A software package for solving elliptic partial differential equations, users' guide 6.0*, SIAM, Philadelphia, 1990.

[3] R.E. Bank: *An adaptive, multi-level method for elliptic boundary value problems*, Computing 26, 1981, 91–105.

[4] R.E. Bank, A.H. Sherman, A. Weiser: *Refiment algorithms and data structures for regular local mesh refinement*, in Scientific computing (R. Stepleman, ed.), North-Holland, Amsterdam (1983), pp. 3–17.

[5] R.E. Bank: *A posteriori error estimates, adaptive local mesh refinement and multigrid iteration*, in Multigrid Methods II, Lecture Notes in Mathematics 1228, Springer, Heidelberg, 1986, pp. 7–23.

[6] P. Bastian: *Parallele adaptive Mehrgitterverfahren.* Teubner, Stuttgart 1996.

[7] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuß, H. Rentz-Reichert and C. Wieners, *UG - A flexible software toolbox for solving partial differential equations*, Computing and Visualization in Science 1, 1997, 27–40.

[8] P. Bastian, *Load balancing for adaptive multigrid methods*, SIAM J. Sci. Comput. 19, 1998, 1303–1321.

[9] P. Bastian, K. Birken, K. Johannsen, S. Lang, V. Reichenberger, G. Wittum, C. Wrobel, *A parallel software-platform for solving problems of partial differential equations using unstructured grids and adaptive multigrid methods*, In: High performance computing in science and engineering (E. Krause and W. Jäger, eds.), pp. 326–339, Springer, Berlin, 1999.

[10] P. Bastian, S. Lang *Complex benchmark computations with UG*, Technical Report 2002-31, IWR (SFB 359), University of Heidelberg, 2002.

[11] J. Bey: *Tetrahedral grid refinement*, Computing 55, 1995, 355–378.

[12] J. Bey: *Finite-Volumen- und Mehrgitterverfahren für elliptische Randwertprobleme.* Dissertation, Universität Tübingen, 1997.

[13] J. Bey: *Simplicial grid refinement: on Freudenthal's algorithm and the optimal number of congruence classes*, Numer. Math. 85, 2000, pp. 1–29.

[14] K. Birken: *Dynamic Distributed Data (DDD) — a software tool for distributed memory parallelization.* Dokumentation, 1997.
`http://cox.iwr.uni-heidelberg.de/~ddd/`

[15] K. Birken, *Ein Modell zur effizienten Parallelisierung von Algorithmen auf komplexen, dynamischen Datenstrukturen*, Ph.D. thesis, University of Stuttgart, 1998.

[16] B. L. Chamberlain: *Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations.* Technical report, University of Washington, 1998.
`http://www.cs.washington.edu/homes/brad/cv/pubs/degree/generals.html`

[17] H. Freudenthal: *Simplizialzerlegungen von beschränkter Flachheit*, Annals of Mathematics 43, 1942, pp. 580–582.

[18] S. Groß, J. Peters, V. Reichelt, A. Reusken: *The DROPS Package for Numerical Simulations of Incompressible Flows Using Parallel Adaptive Multigrid Techniques.* IGPM Report 211, 2002.
`http://www.igpm.rwth-aachen.de/reports/html/rep_2002.html`

[19] S. Groß: *Parallelisierung eines adaptiven Verfahrens zur numerischen Lösung partieller Differentialgleichungen*, Master thesis, in German, RWTH Aachen, 2002.
`http://www-igpm.rwth-aachen.de/www/diplomarbeiten.html`

[20] G. Haase: *Parallelisierung numerischer Algorithmen für partielle Differentialgleichungen.* Teubner, Stuttgart; Leipzig 1999.

[21] G. Karypis, K. Schloegel, V. Kumar: *ParMETIS. Parallel Graph Partitioning and Sparse Matrix Ordering Library.* Dokumentation, University of Minnesota, Army HPC Research Center Minneapolis, 1998.
`http://www-users.cs.umn.edu/~karypis/metis/parmetis/`

[22] G. Karypis, K. Schloegel, V. Kumar: *Parallel Static and Dynamic Multi-Constraint Graph Partitioning.* University of Minnesota, Supercomputing Institute, 2001.

[23] KASKADE. A toolbox for adaptive multilevel codes.
`http://www.zib.de/Scisoft/kaskade2/`

[24] S. Lang, Parallele numerische Simulation instationärer Probleme mit adaptiven Methoden auf unstrukturierten Gittern, Ph.D. thesis, University of Stuttgart, 2001.

[25] S. Lang, *UG - A parallel software tool for unstructured adaptive multigrids*, In: Parallel

Computing: Fundamentals & Applications, Proceedings of the International conference ParCo'99 (E.H. D'Hollander, J.R. Joubert, F.J. Peters, H. Sips, eds.), Imperial College Press, Delft, 1999.

[26] A. Liu, B. Joe: *Quality local refinement of tetrahedral meshes based on bisection*, SIAM J. Sci. Comput. 16, 1995, 1269–1291.

[27] A. Liu, B. Joe: *Quality local refinement of tetrahedral meshes based on 8-subtetrahedron subdivision*, Math. Comp. 65, 1996, 1183–1200.

[28] J.M.L. Maubach: *Local bisection refinement for N-simplicial grids generated by reflection*, SIAM J. Sci. Comput. 16, 1995, 210–227.

[29] MG. A parallel multilevel platform for unstructured grids. `http://rcswww.urz.tu-dresden.de/~jstiller/projects/mg/`

[30] W.F. Mitchell: *Adaptive refinement for arbitrary finite-element spaces with hierarchical basis*, J. Comput. Appl. Math. 36, 1991, 65–78.

[31] M.C. Rivara: *Algorithms for refining triangular grids suitable for adaptive and multigrid techniques*, International Journal of Numerical Methods in Engineering 20, 1984, 745–756.

[32] C.T. Traxler: *An algorithm for adaptive mesh refinement in n dimensions*, Computing 59, 1997, 115–137.

[33] UG: `http://cox.iwr.uni-heidelberg.de/~ug/`

[34] C. Walshaw, M. Cross, M. Everett: *Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes.* J. Parallel Distrib. Comput. 47 (1997), 102–108.

[35] S. Zhang: *Successive subdivisions of tetrahedra and multigrid methods on tetrahedral meshes*, Houston J. Math. 21, 1995, 541–556.