SIAM J. SCI. COMPUT.
Vol. 39, No. 2, pp. C65–C95 © 2017 Society for Industrial and Applied Mathematics

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

REFFICIENTLIB: AN EFFICIENT LOAD-REBALANCED ADAPTIVE MESH REFINEMENT ALGORITHM FOR HIGH-PERFORMANCE COMPUTATIONAL PHYSICS MESHES*

JOAN BAIGES† AND CAMILO BAYONA†

Abstract. In this paper we present a novel algorithm for adaptive mesh refinement in computational physics meshes in a distributed memory parallel setting. The proposed method is developed for nodally based parallel domain partitions where the nodes of the mesh belong to a single processor, whereas the elements can belong to multiple processors. Some of the main features of the algorithm presented in this paper are its capability of handling multiple types of elements in two and three dimensions (triangular, quadrilateral, tetrahedral, and hexahedral), the small amount of memory required per processor, and the parallel scalability up to thousands of processors. The presented algorithm is also capable of dealing with nonbalanced hierarchical refinement, where multirefinement level jumps are possible between neighbor elements. An algorithm for dealing with load rebalancing is also presented, which allows us to move the hierarchical data structure between processors so that load unbalancing is kept below an acceptable level at all times during the simulation. A particular feature of the proposed algorithm is that arbitrary renumbering algorithms can be used in the load rebalancing step, including both graph partitioning and space-filling renumbering algorithms. The presented algorithm is packed in the Fortran 2003 object oriented library `RefficientLib`, whose interface calls which allow it to be used from any computational physics code are summarized. Finally, numerical experiments illustrating the performance and scalability of the algorithm are presented.

Key words. adaptive mesh refinement, adaptivity, finite elements, finite volumes, finite differences, high-performance computing, parallel, load rebalancing

AMS subject classifications. 68U01, 68U20

DOI. 10.1137/15M105330X

1. Introduction.

Discretized partial differential equations are used to solve many types of practical problems in engineering and physics. In some of these problems, the solution leads to a wide range of spatial scales which spread over the computational domain. In these cases, the numerical solution obtained with coarse meshes is often too inaccurate, but performing computations using fine meshes is impractical considering the required computational effort. Adaptive mesh refinement (AMR) methods deal with this issue by producing efficient meshes that are capable of resolving a wide range of scales. These methods locally adjust the mesh to both improve the solution and minimize the computational effort.

Development of parallel AMR methods is justified in order to solve problems that contain a large number of unknowns and which typically require the use of a huge amount of computational resources. Parallelizing the refinement methods allows us to exploit the calculation capabilities provided by rapidly evolving parallel computer clusters. However, parallelized refinement methods lead to a distributed mesh

*Submitted to the journal's Software and High-Performance Computing section December 17, 2015; accepted for publication (in revised form) November 17, 2016; published electronically March 30, 2017.

http://www.siam.org/journals/sisc/39-2/M105330.html

Funding: This work was partially supported by the Spanish Government Elastic-Flow project DPI2015-67857-R. The first author's work was supported by the Spanish Government through Ramòn y Cajal grant RYC-2015-17367. The second author was supported by the doctoral scholarship received from the Colombian Government-Colciencias.

†Centre Internacional de Mètodes Numèrics a l'Enginyeria (CIMNE), Edifici C1, Campus Nord UPC C/ Gran Capità S/N, 08034 Barcelona, Spain, and Universitat Politècnica de Catalunya, Jordi Girona 1-3, Edifici C1, 08034 Barcelona, Spain (jbaiges@cimne.upc.edu, cbayona@cimne.upc.edu).

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

structure, which is complex because frequent data access is necessary, and memory consumption is high. In addition, the dynamical evolution of information during the AMR constitutes another major challenge: it requires a growing number of collective communication operations, and therefore it is not easily scalable in massive parallel computers. Including the possibility of redistributing the workload between processors in order to maximize the utilization of computational resources significantly increases the communications demand. Hence, efficient algorithms and data structures have become the backbone of parallel AMR methods, and distributed collections of structures that can be dynamically modified without requiring several global communications are the preferred designs.

A first approach to parallel AMR methods was block-structured methods. These methods refine parallel meshes by using a single sequential mapping, and therefore are not suitable for complex geometries and nonstructured meshes. Tree-based methods were an alternative to the regularity imposed by the block-structured methods. Tree data structures, namely quadtrees and octrees, are hierarchical data structures constructed with axis-aligned lines and planes. These data structures are used for searching procedures because their hierarchical structure reduces the complexity of the search. The first application of tree data structure algorithms was in parallel domain decomposition and efficient partitioning of meshes (see, for example, Campbell et al. [7]). Later, data structures, balancing algorithms, and adaptive refinement algorithms over distributed octree meshes were developed in [19, 20]. The *etree* library [22] collected algorithms that addressed operations over an octree-based mesh in a database-oriented framework. The code demonstrated good scalability and parallel efficiency. Furthermore, octree developments were implemented into the *Octor* parallel meshing tool [21], which could generate static unstructured meshes on the processors but also performed dynamic refining during execution time. Scalability tests were addressed up to 62,000 processors using hexahedra and giving overall good performance. Some multigrid solvers exploited the balancing and meshing algorithms for octree-based meshes and were implemented in *Dendro* software [17]. The code was scaled up to thousands of processors. Other applications and multiple implementations based on octree data structures were developed in [16, 15] and possessed good adaptivity and performance.

Instead of the quadrilateral and cube-shaped domains that were described by tree data structures, a wider variety of geometries were described by forest-of-octrees–based meshes. This approach was first introduced into AMR methods with the *deal.II* software [3], but the code replicated the global mesh into all processors, and hence it limited the scalability to a few processors. Fully distributed algorithms handling forest-of-octrees meshes were the following step. Burstedde et al. [5] worked in a dynamic AMR based on distributed forest-of-octrees geometries. This was the first work that supported high-order discretizations and non-Cartesian geometries, and it led to the encapsulation of algorithms into the *p4est* library [6]. Good, strong, and weak scaling results over 224,000 cores were obtained for *p4est* working as a parallel adaptive refinement library on meshes composed of quadrilateral and hexahedral elements [3]. Later, Isaac, Burstedde, and Ghattas [11] focused on the balance structure and proposed a subtree balancing algorithm. Weak scaling times improved and required less memory than previous balance algorithms in *p4est*.

In this paper we describe a general adaptive finite element framework for unstructured meshes that has demonstrated suitable performance for large scale parallel computations. The algorithm currently focuses on h-refinement; the extension of the algorithm to hp-refinement will be a matter of future work. Contrary to other paral-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

lel refinement algorithms, the method we present here is developed for nodally based parallel domain partitions; that is, the nodes of the mesh belong to a single processor, whereas elements can belong to multiple processors if they own nodes belonging to different subdomains. These remote nodes over the set of overlapping elements are called "ghost" points. This poses some challenges in parallel communications, since neighboring parallel domains need to be kept updated. Hence, local elements, points, edges, faces, and connectivities are stored in data structures that can be easily accessed and modified. Refinement operations and load balancing procedures are handled over these structures.

To the best of our knowledge, `LibMesh` [14] was a similar approximation. However, because completely unstructured methods work at the cost of having to store explicitly the connectivities of the mesh, the parallel partitioning scheme of `LibMesh` stored all of the mesh information in each processor, and the associated overhead limited the scalability to 100 processors. Jansson, Hoffman, and Jansson [12] also implemented a general adaptive finite element framework for unstructured tetrahedral meshes without hanging nodes, which is suitable for large scale parallel computations. These authors presented strong scaling results linear up to 1,000 processors for an incompressible flow solver. In contrast, the main contributions of the proposed refinement framework are the following:

1. A hierarchical adaptive refinement algorithm for nodally based partitions in distributed memory machines is presented. The algorithm allows us to successively refine and unrefine computational meshes in order to adapt to the requirements of the simulation.
2. Our distributed structure handles two- and three-dimensional unstructured meshes composed of triangular, quadrilateral, tetrahedral, and hexahedral elements. This approach is capable of describing complex geometries and performing nonuniform refinements.
3. We propose a distributed scheme in which each processor stores only the local information of the partitioned distributed mesh. This reduces the memory consumption and allows scaling up to thousands of processors.
4. Our parallel refinement procedure is based on a hierarchical data structure for the refined elements of the mesh that we use to efficiently search neighboring elements at the interprocessor level. A data structure containing parent and children pointers is used, where new refinement levels are successively added to or subtracted from the computational mesh.
5. Resulting meshes are nonconforming with *hanging nodes* on sides where two levels of refinement meet. Contrary to other adaptive refinement methods, the algorithm proposed here does not enforce a *balancing* restriction in the refinement level of adjacent elements: the jump in the refinement level between neighbor elements can be arbitrarily large.
6. For the parallel refinement process, the proposed algorithm deals with element and node identification across processors by using a global element and global point identifier structure. This ensures that the global numbering structure and general nodal and elemental information can be transferred to all of the neighboring processors in an efficient manner.
7. To balance the processors' load, we use a dynamical parallel repartitioning framework that changes the ownership of the mesh nodes when load unbalance reaches a certain threshold, and then it transfers the associated elements to the corresponding processors. Contrary to other algorithms for load rebalancing in hierarchical AMR, the algorithm we propose is independent from

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

the renumbering strategy of the load rebalancing process. In particular, both graph partitioning schemes and space-filling methods for load rebalancing can be used with the proposed algorithm.

The proposed algorithms are packed in an adaptive refinement library which we call `RefficientLib`. The calls to the library have been made as simple as possible so that it can be easily coupled with existing finite element, volume, or difference codes.

Several numerical tests are carried out in order to assess the performance of the proposed methods. The first group of tests corresponds to simulation driven experiments which illustrate the capability of the method to generate computational meshes for different physical problems. A Poisson heat transfer problem is solved both for bidimensional and three-dimensional elements. The incompressible flow past a cylinder is also tested in order to apply the AMR to the incompressible Navier–Stokes equations. In the second group of experiments, weak scalability tests for uniform refinement and load balancing cases in a high-performance computing environment are presented.

The paper is organized as follows. In section 2 the distributed refinement structure with the mesh partition strategy, the distributed data structures, and the initialization of the refinement procedure, are described. In section 3 the refinement step is described. Classification, local refinement, hanging nodes, and exportation to the external flat mesh algorithms are presented. Load rebalancing and global renumbering procedures are included in section 4. The external calls and the user interface to the `RefficientLib` library from an external computational physics solver are presented in section 5. Numerical experiments and scalability tests are presented in section 6. Finally, in section 7 some conclusions are stated.

2. Distributed refinement structure.

In this section we describe the distributed structure of the AMR method. The domain partition strategy is explained first, over which the parallelization is developed. Then we introduce the main data structures and the initialization steps of the parallelized AMR method.

2.1. Mesh partition.

The algorithm described in this paper is designed to work in distributed memory parallel machines. The idea is to have a library which takes care of all the steps necessary for the refinement, while the external driver (for instance, a finite element solver) sees the resulting mesh as a nonhierarchical or flat grid. The domain partition strategy for the mesh is nodal based, which means that each node is assigned to a unique processor, but elements can belong to multiple processors if they own nodes from more than one subdomain. The concepts *point* and *node* both refer to nodes of the mesh, although *node* will generally be used when dealing with points in an element, and *point* will be used when treating them as independent entities. Points belonging to a given subdomain are denoted as *local points*. Before refinement, nodes are assigned to a single processor, but the first layer of nodes belonging to a neighbor processor is also stored in the current processor. These neighboring points are called *ghost* points. Elements can belong to multiple processors if they have nodes from multiple subdomains. We define the *processor responsible for an element* as the processor which owns the node of the element with the lowest global node number.

Figure 1 shows an initial domain partitioned mesh as seen from different processors. The advantage of this strategy is that each processor stores only the local information of its subdomain. The processor stores the local numeration of the subdomain points, but the global numbering of these points must also be saved in order to locate and communicate points for other processors.

The parallel refinement method is constructed over the partitioned mesh. Since the mesh needs to be seen as a flat mesh by the external driver, a node and element

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

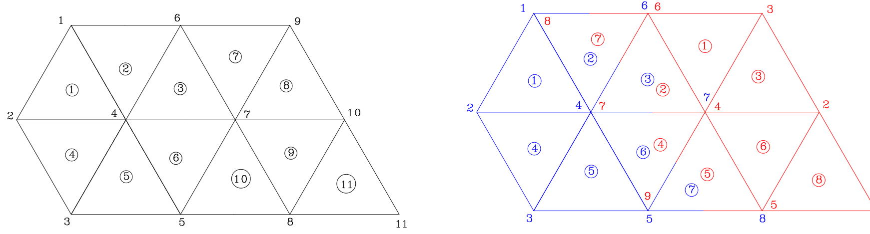

FIG. 1. *Initial mesh information. Left: the global mesh. Points and elements (circled) are numbered globally. Right: the mesh, partitioned into two subdomains. Colors denote the processor to which the information belongs. Points and elements (circled) are numbered locally for each domain. Note that the elements numbered* (2), (3), (6), *and* (10) *in the global mesh are shared by the two subdomains. This distributed structure will be used to calculate remote neighboring contributions for the shared elements. (See online version for color.)*

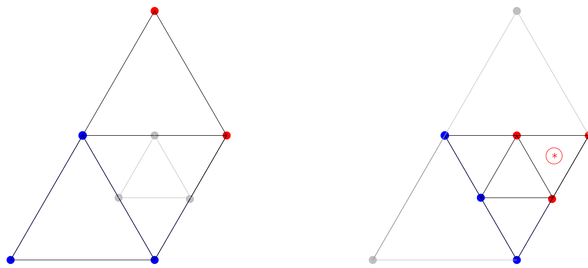

FIG. 2. *Internal hierarchical mesh as seen from one of the processors. In order to illustrate this refinement example, a shared element of the mesh of Figure* 1 *is refined. Left: initial level* 0 *mesh. Right: refined level* 1 *mesh.*

renumbering strategy is needed in order to be able to move from the *external*, flat mesh to the *internal* (refiner) hierarchical mesh and vice versa. Figure 2 presents an example of a hierarchically refined mesh. Two levels of refinement are displayed as seen internally in the refiner from one of the processors. The same mesh is depicted in Figure 3 as seen from the external driver. Note that the element marked with an asterisk ($*$) in Figure 2 does not appear in the flat mesh for the considered processor: in the external flat mesh, only the first layer of flat node neighbors is considered, while in the internal hierarchical mesh, also the element hierarchical neighbors are considered. Element hierarchical neighbors are elements which share a parent with an element that belongs to a processor. The element marked with an asterisk ($*$) in Figure 2 is a hierarchical neighbor, because although none of its nodes is assigned to the processor, the element shares its parent with the rest of level 1 elements in the processor. We call the elements which share a parent *sibling elements*.

2.2. Distributed data structures. The algorithms to be described in section 3 are implemented by using a collection of data structures which allow us to efficiently

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

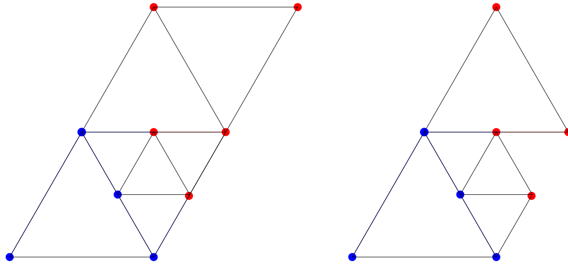

Fig. 3. *Refined mesh as seen from the external driver. Left: external flat mesh. Right: external flat mesh as seen from the processor denoted by the blue color. (See online version for color.)*

access and modify the information that defines the mesh. The implementation is generally done in an object-oriented manner, but for efficiency the actual storage in memory sometimes uses a *flattened* storage, where parts of the objects are stored in an array list. We will denote this storage structure as *CSR*, in reference to the *compact sparse row* storage used in many computational physics applications. All of the data structures are encapsulated in a class, which we call the **Refiner**. Refinement procedures and communications are performed by this class. The data structures which are part of the Refiner class are briefly explained as follows:

- **gnpoin** is the total number of points of the internal mesh.
- **gnelem** is the total number of elements of the internal mesh.
- **npoin** is the number of points of the internal mesh in the local processor. This includes both the local (belonging to the current processor) points (**npoinLocal**) and the first layer of hierarchical neighbors of the local points (**npoinGhost**).
- **npoinLocal** is the number of points of the internal global mesh belonging to the current processor.
- **npoinGhost** is the number of points of the internal global mesh which are the first layer of hierarchical neighbors of the local nodes. Their data are required in the current processor.
- **nelem** is the number of elements of the internal mesh which are required in the current processor. This includes not only all elements having local nodes but also elements which do not have a local node but are relevant in the hierarchical refinement process.
- **ElementList** is the list of elements in local numbering, of size **nelem**. For each local element we need to store it contains
  - **ElementType.** This item identifies the element type (triangles, quadrilaterals, tetrahedra, hexahedra) and subtype or variation according to the subdivision process. The subtype is relevant, for instance, in the case of tetrahedral elements, where there are multiple possibilities for hierarchically subdividing an element. This is stored as an array of 1 byte integers of dimension 2.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

  - Data structure of type **GlobalElementIdentifier**. Similarly to the strategy followed in [3], the global element identifier allows us to uniquely identify an element from the mesh (described in subsection 2.3).
  - **ParentIdentifier.** This is the local element numbering of the parent element.
  - **ChildrenIdentifierList.** This is the list of local element numbering of the children elements. (Stored in *CSR* format.)
  - **NodeList.** This contains the nodes which are part of the element in local numbering. This is stored as an array of 4 byte integers of dimension 2. (Stored in *CSR* format.)
  - **FaceList.** For each face in the element, it stores the neighbor (opposite) element and the corresponding face (or edge in two dimensions) of the neighbor. If this is a hanging face, it contains the neighbor of the parent element and the corresponding face. (Stored in *CSR* format.)
- **PointList** is the list of nodes in local numbering, of size **npoin**. The first **npoinLocal** components of the list correspond to local points, and the last **npoinGhost** components of the list correspond to ghost points. For each point, we store the following:
  - **GlobalPointNumbering.** This data structure stores the global (parallel) point numbering of the point. An **InverseGlobalPointNumberingList** is also created which allows us to get the local point number of global points for a given processor. In order to implement the inverse global point numbering list, a hash table type structure is used (this is described in subsection 2.4).
  - **ProcessorNumber.** For ghost points, it stores the processor number to which the point belongs.
  - **Level.** This is the level (in the refinement structure) of the point. Initial points are classified as level 0.
  - **EdgeRefinementList.** For each point i, it stores a list of neighbor points j to which point i is connected only if a refinement node k between i and j exists; it also stores the local numbering of j and k. (Stored in *CSR* format.)
  - **HangingNodeList.** For hanging nodes, it contains the list of recursive parent nodes and their linear combination coefficients. (Stored in *CSR* format.)

Most of the lists involving multiple types of data (i.e., **ElementList** or **PointList)** are implemented as separated list arrays for convenience and memory performance, although they could also be stored as objects containing data structures. When the size of each component of the list is not constant for all elements, these separated list arrays are generally stored in *CSR* format.

2.3. The GlobalElementIdentifier data structure. The hierarchical refinement element information is encapsulated into a tree data structure composed of **GlobalElementIdentifier** objects. This data structure allows us to uniquely identify an element in any processor, at any level of refinement. When used together with the **ParentIdentifier** and **ChildrenIdentifierList**, it allows us to communicate element information between processors. This information can be used to identify the local numbering of an element when interprocessor information communication is required. The **GlobalElementIdentifier** structure is composed of the following information:

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

- **GlobalTopLevelElement.** This is the original element number of the top level element prior to any refinement or load rebalancing process. This is stored as a 4 byte integer.
- **Level.** This is the level of the element in the refinement structure. Initial elements are classified as level 0. This is stored as a 1 byte integer.
- **PositionInParentElement.** For each level, three bits are dedicated to store the refinement branch (or child) of the element. This allows us to identify a maximum of eight children per level. A maximum of 21 refinement levels is allowed in the current implementation, totalling 63 bits. This is stored as an 8 byte integer.

The total amount of memory required to store the **GlobalElementIdentifier** for an element is 13 bytes, which round up to 16 bytes in memory.

2.4. The InverseGlobalPointNumberingList. In order to perform parallel communications, we need to be able to recover the local point number from a global point identifier at any time in the refinement process. This could be done in a straightforward manner by allocating an array of dimension **gnpoin** and then storing for each local point, in the global position of the array, the local number associated to the corresponding global point. However, **gnpoin** depends on the size of the global problem and can in general be very large. This would result in the allocation of this array becoming very time consuming and, when using thousands of processors, affecting the performance and scalability of the parallel refinement algorithm.

Thus, an alternative implementation of the global to local mapping is required. In our implementation, we have opted to implement it by using a hash filtering followed by a binary search if collisions are found.

- First, the hash function is defined as the modulus of the division by a primer number **primeNumber,** which is close to and larger than the local number of points **npoinLocal**. A hash table data structure of dimension **primeNumber** is allocated.
- Second, for each local point–global point pair, the hash function of the global point identifier is computed and the point is stored in the corresponding hash table data structure slot. If there are collisions, the points are stored in ascending order according to their global point identifier.
- Finally, in order to recover the local numbering of a global point, the hash function of the global point number is computed. If a single point is stored in the corresponding hash table slot, the local numbering is recovered directly. In case there are collisions, a binary search is performed on the sorted global point numbering array of the hash table slot in order to find the corresponding local point number.

This process allows us to reduce the average computational cost of finding the local point number associated to a global point to $\mathcal{O}(1)$, while keeping the storage requirements to $\mathcal{O}(\textbf{npoinLocal})$. Although a possibly more efficient implementation could be found by differentiating between the behavior of the algorithm for local and ghost points, we have chosen the described implementation for its compromise between efficiency and reusability.

2.5. Initialization. Our parallel refinement method establishes the input initial mesh as a zero level mesh that cannot be coarsened. In the initialization step, the flat, top level mesh is passed to the **Refiner**. From this mesh, **ElementList** and **PointList** are built. All elements and points are assigned the zero level. The processor number for each point and a global element number for each element and

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

point need to be passed to the **Refiner**, from which the **GlobalElementIdentifier** for each element and the **GlobalPointNumbering** (and its inverse) for each point can be built. The initial **FaceList** for each element is built by looping through neighbor elements and checking for faces with coincident nodes. Neighbor elements are identified in a two step process as elements with which at least one point is shared. The remaining arrays, which refer to the refinement structure, are started as empty or null, since no refinement step has been performed yet.

3. Refinement step.

3.1. Amending the element refinement classification.

The first stage of the refinement step consists of passing to the library an array of size **nelem** which contains the information about the element refinement. In our implementation this is achieved by passing a 1 valued integer for elements which need to be refined, a -1 valued integer for elements which need to be unrefined, and a 0 valued integer for elements which do not need to be either refined or unrefined. Elements can only be unrefined if all of their sibling elements are also unrefined. If an element is marked to be unrefined but one of its siblings is not, then the element is reclassified as not to be unrefined.

An important point is that the refinement criteria must be the same in all processors; that is, if an element belongs to both processors i and j, then the decision of whether to refine the element must coincide in processor i and processor j. Even when taking this into account, there are unrefinement cases in which the information available in a certain processor is not enough to decide whether an element will effectively be unrefined. This is, for instance, the situation for an unrefinement step of the level 1 elements in Figures 2 and 3. Suppose that the external driver has marked the four level 1 elements which are exported to the external mesh (Figure 3) to be unrefined. However, the blue processor (Figure 3, right) has no information on the classification of the hierarchical neighbor (marked with an asterisk ($*$) in Figure 2). As a consequence, and only in the case when all the local element siblings are marked to be unrefined, a communication step with the neighbor processor is required in order to classify hierarchical neighbor elements.

This communication step consists of asking the processor responsible for the element to communicate its refinement classification. Elements in different processors are identified through their **GlobalElementIdentifier**. This step is not required when refining, because sibling elements can be refined independently.

As explained previously, the proposed algorithm can deal with unbalanced meshes in the sense that the refinement level jump between neighbor elements can be arbitrarily large. However, this might not be convenient for some applications. For this, an optional flag which limits the level jump between neighbor elements has been added to the algorithm. If this flag is enabled, the algorithm adds the following to the previous reclassification of elements to be refined: for each node, it notes the maximum and minimum levels of the elements to which it belongs. Then, if the difference between the maximum and minimum levels is 1 (the algorithm should not allow this difference to be larger than 1), it does not allow the maximum level elements to further refine, and it does not allow the minimum level elements to unrefine. This ensures that a balanced mesh is obtained in the case when this flag is enabled.

3.2. Local refinement.

Once all elements are properly classified following the refinement criteria, a local refinement step starts in each processor. The implemented subdivision process for triangles, quadrilaterals, tetrahedrons, and hexahedrons re-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

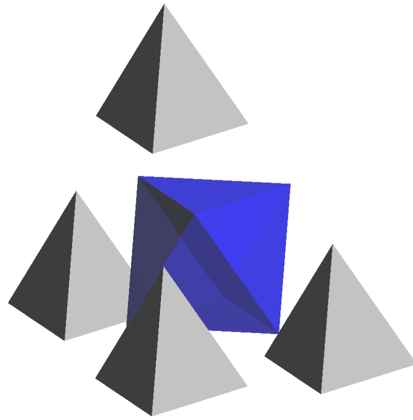

Fig. 4. *Subdivision of a tetrahedron into four tetrahedrons and one octahedron. The octahedron is then further subdivided into four tetrahedrons.*

fines a given element into 2^d subelements of the same type, where d is the number of dimensions, although the implementation is left open to refine into other element types in the future. A first loop through the elements allows us to compute the dimensions of the arrays after the refinement stage, which are then allocated. Elements which are unrefined are removed from the lists, and new elements are added at the end of the **ElementList**. At the same time, the **ParentIdentifier** and the **ChildrenIdentifierList** for each element are filled. Also the **GlobalElementIdentifier** for each new element is computed from the **GlobalElementIdentifier** of its parent element (adding one level to the parent level, assigning a child number for the new level). Also the element type is computed (for h-refinement, the element type of children elements is the parent element type). In the case of tetrahedrons, what we call an element subtype also needs to be stored: each refined tetrahedron is subdivided into eight subelements. However, there are three different ways of subdividing a tetrahedron into eight subelements, each way corresponding to a main plane of subdivision of the internal octahedron obtained by joining the midpoints of tetrahedron edges. The election of the subelement type is done so that the distortion of the resulting children elements is minimized. This is illustrated in Figure 4.

For updating the **FaceList** of each element, the neighbors of each element are checked. If in the previous refinement step the face was connected to an element which has been unrefined in the current refinement step, then the element is connected to the parent element. On the contrary, if the element is connected to a higher level element which has now been refined, then the element becomes connected with the corresponding children. These face connections can be done in an efficient manner thanks to the **Parent-Identifier** and **ChildrenIdentifierList** structures, which allow us to move through the different refinement levels. Some examples of face matching for elements in different levels are shown in Figure 5. Faces that connect to faces in elements of a higher level are denoted as *hanging faces*.

At this point, the element refinement structure has been updated to the new refinement stage. However, the new nodes still need to be added to the new elements. There are three types of nodes in the new elements: nodes which are added in the edges

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

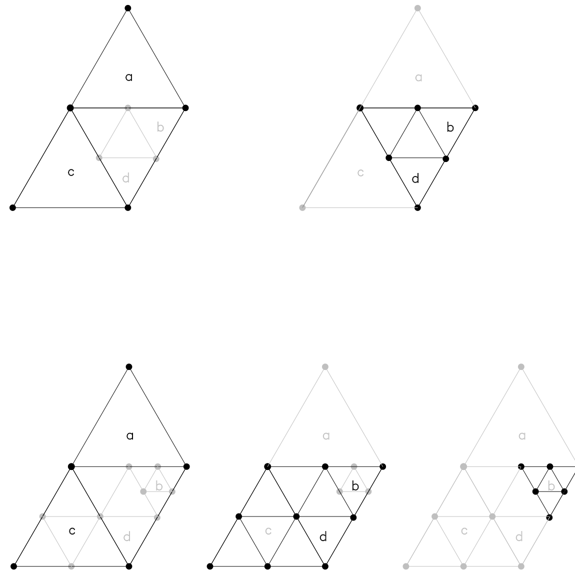

Fig. 5. *Face matching in the refinement process. In the mesh at the start of the refinement process (top), element* b *has a face connected to element* a*, which is one level higher. Similarly, element* d *has a face connected to one of the faces of element* c*. After a refinement step, both elements* c *and* b *are refined (bottom). Some of the children of* b *have a face which is connected to a face in element* a*, while the faces in element* d *which were connected to element* c *are now connected to faces of the children of element* c*.*

of the parent element, nodes which are added in the faces of the parent element, and nodes which are added in the interior of the parent element. This is best illustrated in three-dimensional hexahedral elements, as shown in Figure 6.

For nodes added in the interior of the parent element, we are sure that the node is new, and a new point number can be assigned to the node. However, for nodes added in the faces or in the edges of a parent element, we need to check that the node does not already exist in another element.

In the case of nodes added in the faces of elements, the node will pre-exist only if the face of the new child element is connected to an element of the same level. In this case the new node in the new child element is assigned the point number of the node in the neighbor face.

For nodes added in the edges of elements, we make use of the **EdgeRefinementList** structure, in which for each edge connection between two points of the mesh we store the point numbering of the refined point added between them. This keeps track of the already existing refined points in the edges, allowing us to add new points when an element is refined or, on the contrary, to match existing points with the refined point in the edge. The addition of new nodes to the **EdgeRefinementList** needs to be done in a two step process (the first loop for counting sizes and allocating, and the second loop for filling the data structures) and making use of linked lists in order to obtain a proper performance of the algorithm.

3.3. Parallel numbering. The algorithmic steps in the previous subsection allow us to advance to the new refined mesh. However, no communications have been done between processors, so at this stage new points have been assigned a local

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

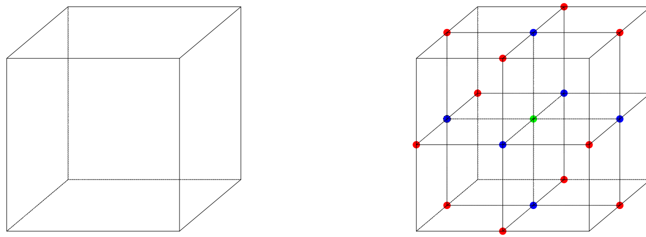

Fig. 6. *Edge (red), face (blue), and interior (green) new nodes in a refined element. (See online version for color.)*

numbering, but the parallel numbering of points is still pending. Elements, on the other hand, are already identified by their **GlobalElementIdentifier**. In order to construct the new global point numbering, we start by classifying points as local or ghost in each processor. Points that already existed in the previous refinement stage keep the same local/ghost status. New points are classified as local or ghost following different criteria depending on whether they are new interior points, new face points, or new edge points.

The new interior points are classified as local for a given processor if, in the previous refinement step, the processor was the owner of the node in the parent element with the lowest global numbering (the processor was the *responsible processor for the element)*.

The new face points are classified as local for a given processor if, in the previous refinement step, the processor was the owner of the node in the parent face with the lowest global numbering (the processor was the *responsible processor for the face)*.

The new edge points are classified as local for a given processor if, in the previous refinement step, the processor was the owner of the node in the parent edge with the lowest global numbering (the processor was the *responsible processor for the edge)*.

Once all points of each processor have been classified as local or ghost, a gather operation of the number of local points of each processor, followed by a scatter operation with the first global point number of each processor, allows us to set the parallel global numbering for all the local points in each processor (since the global numbering of the local points of each processor will be consecutive). However, the parallel numbering of the ghost points of each processor is still unknown to the processor. For points that were already ghosts in the previous refinement step, each processor simply asks the owner of the point to communicate its new global number. When asking for the point, this point is identified by its global number in the previous refinement step.

For new ghost points, their global numbering is obtained as follows:

- For new interior and face points, the **GlobalElementIdentifier** of the parent element is sent to the responsible processor together with the interior/face node number, which in turn returns the global point number for the interior point.
- For new edge points, the old (previous refinement step) global numbering of the edge parent points is sent to the processor responsible for the edge, which in turn seeks the refined edge point through its **EdgeRefinementList** and returns the new point global number for the new edge point.

At this stage, all internal refinement structures are already defined and updated to

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

the new refinement stage. The steps for exporting this information to the external mesh are detailed in the following subsections.

3.4. Hanging nodes. Although in most refinement algorithms using tree data structures a conforming restriction over adjacent elements must be satisfied, this is not the case in our algorithm. This so-called balance condition, which enforces the fact that there is only one *hanging node* on sides where two levels of refinement meet, does not need to be satisfied by the algorithm presented in this paper, and an arbitrary jump in the refinement level of adjacent elements is possible. We believe that this is one of the main features of the present algorithm.

Hanging nodes are classified as nodes which belong to a face or edge that is connected to an element in a higher level (hanging face or edge) and which do not belong to the higher level element. There are several possibilities for treating hanging nodes in computational physics: one of the possible approximations for hanging nodes consists of fixing the value for the unknowns in the hanging node as the mean of the value of the unknowns of its hanging parents (this is the approach we have followed in the numerical examples). Other possibilities include the use of discontinuous Galerkin methods [8] or hybrid continuous-discontinuous Galerkin methods [2].

In any case, it is necessary to know which are the hanging parents of a certain hanging node. Hanging parents are defined as nodes in the parent element in the case of interior refined nodes, nodes in the parent face in the case of face refined nodes, and nodes in the parent edge in the case of edge refined nodes. If the hanging parents are in turn hanging nodes, this establishes a recursive dependency of the values of the unknowns in the hanging nodes on the values of the unknowns in higher level nodes.

In our implementation, the list of hanging nodes and their relation with respect to their hanging parents is obtained by looping through the elements and checking the node matching of faces and edges which connect to higher level elements. The nodes that are not present in both sides of the face, or in all elements which concur at the edge, are considered hanging nodes.

Once the hanging nodes list is built, the recursive dependency structure with respect to their hanging parents is obtained by traversing the hanging nodes list from top level to low level hanging nodes and annotating the contribution of hanging parents to the averaged value of each hanging node. As the number of parents contributing to the averaged value of a hanging node can be arbitrarily large if no balance condition is applied, this recursive dependency structure is stored in *CSR* format. If a parent node of a hanging node is also a hanging node, then its contribution to the value of the hanging node is transferred recursively to the hanging parents. This is illustrated in Figure 7.

Note also that in successive refinement steps, normal nodes can become hanging nodes depending on the refinement behavior of neighbor elements and vice versa.

3.5. Exporting the external mesh. As explained previously, the hierarchical mesh with which the algorithm works internally does not coincide with the flat mesh which is exported and used by the external driver. The main criteria for choosing elements and nodes which are passed to the external mesh are the following:

- Elements are exported only if they are last level elements which own at least a local node.
- Nodes are exported only if they belong to an exported element.

These two criteria are in general sufficient for deciding which elements need to be exported. However, there are some specific cases (after load rebalancing has occurred and children elements are not necessarily in the same processor as their parent ele-

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

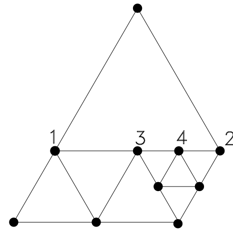

Fig. 7. *Hanging nodes. Node* 4 *is a hanging node, its hanging parents being nodes* 2 *and* 3*. Node* 3 *is in turn a hanging node, its hanging parents being nodes* 1 *and* 2*. The hanging node value for the unknown at node* 4 *is* $u_4 = \frac{1}{2}u_3 + \frac{1}{2}u_2 = \frac{1}{4}u_1 + \frac{3}{4}u_2$.

ments) where additional elements need to be exported. The first is the case of hanging nodes whose hanging parent nodes are assigned to a different processor, which is illustrated in Figure 8. In this case, the elements owning the recursive hanging parents need also to be exported in order to ensure that the assembly to the parent nodes can be performed and to ensure that if the high level element is refined, all of the involved processors will be aware of the refinement step.

The second is the case of elements which are *hanging opposites* of elements with nodes assigned to the current processor. Even if none of the nodes of the element belong to the current processor, these elements need to be exported so that their refinement criteria can be known by the current processor and the new elements and face matching can be created. Again, an example of this particular case can be seen in Figure 8.

In both cases, information on elements belonging to neighbor processors but not present in the current processor will have to be communicated between processors. This is the case of element b in Figure 8.

4. Load rebalancing. After several refinement steps, computational load in the processors may become unbalanced, causing the most loaded processor to damage the global efficiency. In order to avoid this issue, load rebalancing is required. Load rebalancing consists of changing the processor owning each group of nodes in such a way that the computational load is approximately equal in all processors. At the same time, it has to be ensured that the communications required in the load rebalancing process and in the external computations to be performed by the driver are minimized.

4.1. Rebalance renumbering. The first step of the load rebalancing process consists of computing the new processor and new global number for each node in the mesh. Contrary to other adaptive refinement schemes in which the new node numbering is linked to the adaptive refinement algorithm, in the algorithm presented in this paper any node renumbering strategy is possible. In fact, the new node numbering is computed externally by the driver and then passed to the refinement library. In the numerical examples presented in section 6, node renumbering is computed externally by using the ParMetis [13] and Zoltan [4] specialized software, which are based on nested bisection, graph partitioning, and space-filling methods. Both of these packages are accessed through an interface provided by the PETSc library [1], which we also use as a linear system solver for the numerical examples in section 6. All of these

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

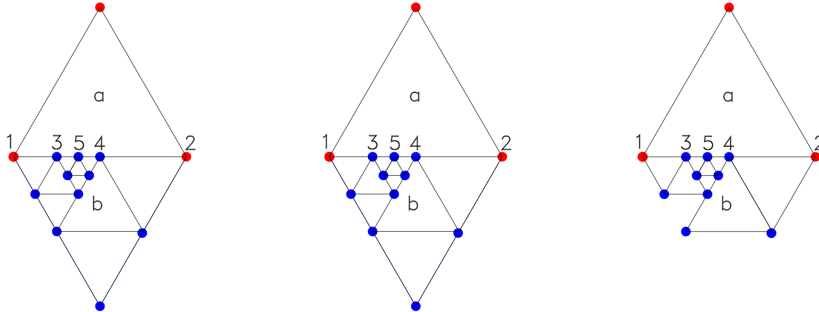

Fig. 8. *Left: global exported mesh. Center: local exported mesh for processor* 0 *(blue nodes). Right: local exported mesh for processor* 1 *(red nodes). Element* a *would in principle not be exported in processor* 0*, since it has no local node for this processor, but it is exported because it is the hanging opposite of elements belonging to processor* 0*. Element* b *would in principle not be exported in processor* 1*, since it has no local node for this processor, but it is exported because it is the hanging opposite of element* a *belonging to processor* 1*. (See online version for color.)*

remapping methods produce new partitions from an already partitioned mesh.

With all the nodes in the mesh already assigned a processor number to which they belong, it remains to decide which elements and nodes need to be sent to each processor. The algorithmic rules of this step are similar to the rules for exporting elements and nodes in subsection 3.5, although taking into account the hierarchical nature of the internal mesh as follows:

- An element needs to be sent to a processor if it owns a node which belongs to the processor.
- An element needs to be sent to a processor if one of its children or sibling elements is sent to the processor.
- A node needs to be sent to a processor if it belongs to at least one element which is sent to the processor.

Again, an additional rule applies in the case of elements with hanging nodes: any element which is the high level hanging (edge or face) opposite of an element which needs to be sent to a given processor will also be sent to the processor. The situation is similar to the one depicted in Figure 8.

4.2. Rebuilding the refinement structure. The final step before the **Refiner** is ready to continue with the following step of the AMR strategy consists of rebuilding all the required data structures. The **FaceList** structure and the **ParentIdentifier** and **ChildrenIdentifierList** arrays can be rebuilt in a two step process from the information in the **NodeList** and by traversing the element levels using the **GlobalElementIdentifier** of each element. Once this is done, rebuilding the **HangingNodeList** and the **EdgeRefinementList** is also straightforward, although some care needs to be taken so that the resulting implementation is efficient.

5. External calls to the `RefficientLib` library.

In this section we present the interface and calls to the `RefficientLib` object-oriented library. The implemen-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

tation allows the refinement algorithm to be used from distributed memory computational physics codes. The library has been developed following the object-oriented Fortran 2003 standard. In the following we illustrate a typical call to the adaptive refinement library.

The first step consists of the initialization of the **Refiner** object. This is done by passing an **IniData** object from which the initial mesh information can be retrieved (elements, nodes, and connectivities). The **IniData** object is defined as abstract in the library and needs to be implemented and extended by the user:

```
!MyIniData is an object from which the initial mesh information
!can be extracted
call Refiner%Initialize(IniData)
```

The required calls for this object are the following:

```
!General dimensions
!Number of points in the processor
call IniData%GetNpoin(npoin)
!Number of local points in the processor
call IniData%GetNpoinLocal(npoinLocal)
!Number of elements in the processor
call IniData%GetNelem(nelem)
!Global Number of Points
call IniData%GetGlobalNpoin(gnpoin)

!Get an element connectivity list
call IniData%GetConnectivity(ielem,nnode,connectivity)

!Get the local to gobal mapping for a set of nnode nodes
!Output is GlobalNumbering
call IniData%GetLocal2Global(nnode,LocalNumbering,GlobalNumbering)

!Get the processor to which a set of nnode nodes is assigned
!Output is ProcessorList
call IniData%GetProcessorNumber(nnode,LocalNumbering,ProcessorList)
```

Also, the coordinates of the nodal points of the mesh are passed to the **Refiner**. These are only used in the tetrahedral elements case in order to choose the subelement type which causes the lesser distortion, as explained previously:

```
!Coord contains the array of coordinates, necessary for the
!subelement type choice in tetrahedral elements
call Refiner%SetCoordArray(coord)
```

An optional call for setting the balancing flag (and forcing the resulting mesh to be balanced) can be done in the following manner:

```
!Optional call for setting the balancing flag
call Refiner%Set_2_1_Balancing(.false.)
```

Once the **Refiner** is initialized, the array containing the refinement criteria is passed to the **Refiner**, which performs all the required refinement steps:

```
if (refining) then
  !RefinerMarkel contains the refinement criteria for each element
   call Refiner%Refine(RefinerMarkel)
endif
```

If instead of a refinement step, we are carrying out a load rebalancing step, an external call to the renumbering library needs to be done, which needs to retrieve the **PointGlobNumber** and the **PointProcNumber** arrays. Once these are available, the **Refiner** call has following form:

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

```
if (rebalancing) then
   !A call to an external library for renumbering is
   ! necessary (user defined)
   call ExternalLibraryRenumbering(PointGlobNumber,PointProcNumber)
   !PointGlobNumber,PointProcNumber contain the new global
   !numbering and processor for points
   call Refiner%SetRebalancingNumbering(PointGlobNumber,PointProcNumber)
   !Communicate and rebuild the refinement structure
   call Refiner%LoadRebalance
endif
```

Once the refinement or load rebalancing step is done by the **Refiner**, information can be retrieved using several calls, and from it the new external flat mesh can be built:

```
!Once the refinement is done, we retrieve the information from the
    !Refiner
!New Dimensions
!number of local points, local + ghost points, global points
call Refiner%GetPointDimensions(newnpoinLocal,newnpoin,newgnpoin)
!number of elements, size of the connectivity array
call Refiner%GetElementDimensions(newnelem,newLnodsSize)

!After allocation of external arrays, the information can
!be retrieved from the Refiner
!Element Connectivity list in CSR format
call Refiner%GetLnods(pnods,lnods)
!Local to Global and processor list for the new local nodes
call Refiner%GetLocalOrdering(LocalToGlobal,ProcessorList)
!A list of hanging nodes (pHangingList, lHangingList)
!with the averaging coefficients (rHangingList)
!in CSR format can be retrieved from the Refiner
call Refiner%GetHangingListDimensions(HangingListSize)
call Refiner%GetHangingList(pHangingList,lHangingList,rHangingList)
!A list of hanging faces (pHangingList, lHangingList)
!in CSR format can be retrieved from the Refiner
call Refiner%GetHangingFacesList(pHangingFacesList,lHangingFacesList)
```

The hanging nodes list contains, for each node, the list of their hanging parents (*lHangingList)* and the corresponding averaging coefficients (*rHangingList)*, stored in *CSR* format.

For all nodal arrays defined in the old mesh, a call to the **Refiner** allows us to transform them (by interpolation and restriction) into arrays in the new mesh:

```
call Refiner%UpdateVariable(ndime,coord,newcoord)
```

This summarizes the interaction with the adaptive refinement library from a user point of view. Some additional optional calls exist that allow us to pass values in the boundaries of the old mesh to the boundaries of the new mesh. These can be convenient, for instance, for enforcing Neumann boundary conditions in finite element analysis, but we have not included them here for legibility and conciseness.

6. Numerical examples.

In this section we illustrate the behavior of the proposed algorithm through several numerical examples. In these examples the algorithm is tested for various types of linear, bilinear, and trilinear elements, in both two and three dimensions.

6.1. Bidimensional elements.

The first numerical example consists of a square bidimensional domain $(0,1)\times(0,1)$ meshed using triangular linear elements. The refinement process is arbitrary in order to show the capability of the algorithm to

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

deal with multilevel jumps across hanging faces, and in this case it concentrates most of the elements in the top-right corner.

The number of processors in this simulation is 25, and the load rebalancing criterion is the following:

$$\frac{\max(\mathbf{npoinLocal}) \cdot \mathbf{nproc}}{\mathbf{gnpoin}} \geq \mathrm{tol}_{\mathrm{rebalance}}.$$

Over this mesh, a Poisson heat transfer problem is solved using finite elements. The heat transfer problem consists of finding $u : \Omega \longrightarrow \mathbb{R}^d$ such that

$$\begin{aligned} -k\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{in } \Gamma_D, \\ k\boldsymbol{n} \cdot \nabla u &= h \quad \text{in } \Gamma_N, \end{aligned}$$

where $k > 0$, f is a given forcing function, h is the normal heat flux, Ω denotes the computational domain, and $\partial\Omega = \Gamma = \Gamma_D \cap \Gamma_N$ is the boundary, with $\Gamma_D \cap \Gamma_N = \emptyset$. In this example the forcing term is uniform with value $f = 1$; Γ_D is composed of the upper, lower, and left boundaries; while Γ_N is composed of the right boundary, with $h = -20$.

Figure 9 shows the behavior of the method for this case. After several refinement/unrefinement steps, the mesh is much more heavily refined in the top-right corner than in the rest of the computational domain. Note that in this process, in some of the steps normal nodes become hanging nodes and vice versa. The load rebalancing acts by rearranging the ownership of the nodes in such a way that the computational load in all of the processors is approximately the same. This results in most of the processors dealing with nodes in the top-right corner. The algorithm is capable of dealing with the multilevel jump in hanging faces and providing an accurate result for the temperature field.

6.2. Multiple types of elements in a single bidimensional simulation. In this numerical example we solve again the example presented in subsection 6.1, but this time we use two types of elements: in the left half of the domain we use triangular finite elements, while in the right half quadrilateral finite elements are used. This example illustrates the capability of the algorithm to deal simultaneously with several types of finite elements. Figure 10 shows the numerical results. Note that some of the hanging faces of the mesh belong to the interface between triangular and quadrilateral elements.

6.3. Tetrahedral and hexahedral elements. In this numerical example a heat transfer problem is solved in a unit cube domain. The boundary conditions are adiabatic in all walls except in the lower one, where the temperature is fixed to zero. The source term is $f = 1$. The selection of the elements to refine is again arbitrary, and several refinement/unrefinement steps are performed before arriving at the final configuration. The number of processors in this numerical example is 6.

Figure 11 shows the behavior of the method for this case. After several refinement/unrefinement steps, the mesh is much more heavily refined in the top-right quarter than in the rest of the computational domain, and the load rebalancing algorithm acts by rearranging the nodes in each processor so that the computational load is similar in all processors. The algorithm is capable of dealing with the multilevel jump in hanging faces and providing an accurate, smooth result for the temperature field.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

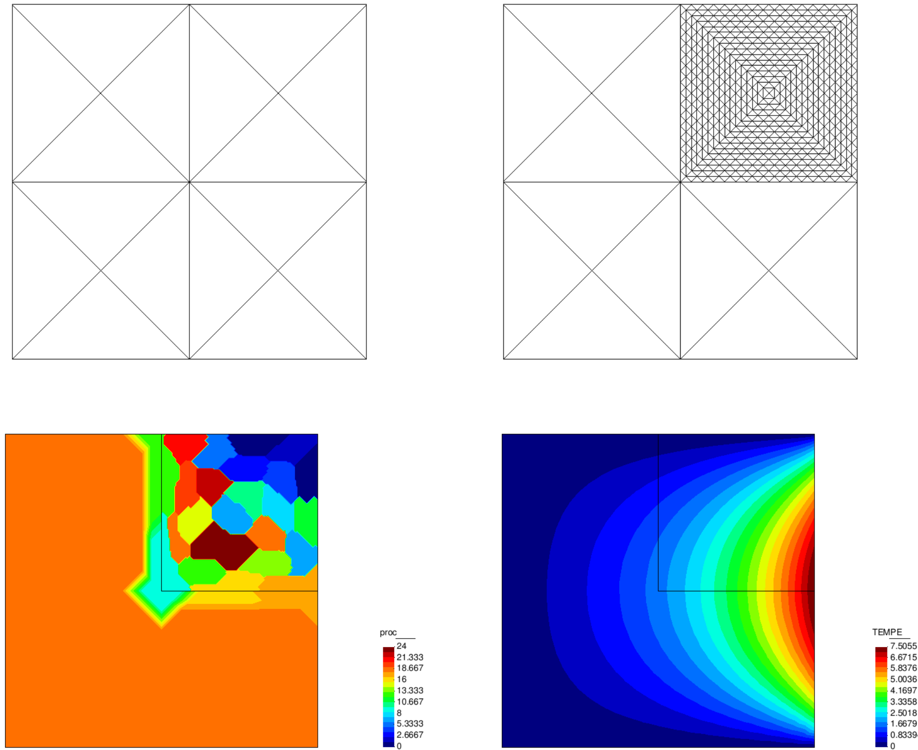

Fig. 9. *Poisson problem example on a triangular mesh. Top-left: original mesh. Top-right: mesh after some refinement steps. Bottom-left: node distribution across processors. Bottom-right: temperature field at the end of the simulation. Note that the jump of refinement level across hanging faces can be arbitrarily large.*

Figure 12 shows the same example using hexahedral elements and 25 processors, the load rebalancing criteria being the same as in the previous cases. In the three-dimensional case no simulation is done using multiple types of elements, since the faces of the elements and the finite element shape functions for tetrahedral and hexahedral elements do not match at the interface.

6.4. An application to the incompressible Navier–Stokes equations. In this numerical example we solve the incompressible Navier–Stokes equations, which consist of finding $\boldsymbol{u} : \Omega \times (0, T) \longrightarrow \mathbb{R}^d$ and $p : \Omega \times (0, T) \longrightarrow \mathbb{R}$ such that

$$\begin{aligned} \partial_t \boldsymbol{u} - \nu \Delta \boldsymbol{u} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} + \nabla p = \boldsymbol{f} \quad & \text{in } \Omega, \\ \nabla \cdot \boldsymbol{u} = 0 \quad & \text{in } \Omega, \\ \boldsymbol{u} = \bar{\boldsymbol{u}} \quad & \text{on } \Gamma \end{aligned}$$

for $t > 0$, where $\partial_t \boldsymbol{u}$ is the local time derivative of the velocity field. $\Omega \subset \mathbb{R}^d$ is a bounded domain, with $d = 2, 3$, ν is the viscosity, and $\boldsymbol{f}$ is the given source term. Appropriate initial conditions have to be appended to this problem. The numerical solution of these equations through finite elements is done using a stabilized formulation [9] which allows us to deal with the convective term and use equal interpolation spaces for the velocity and the pressure.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

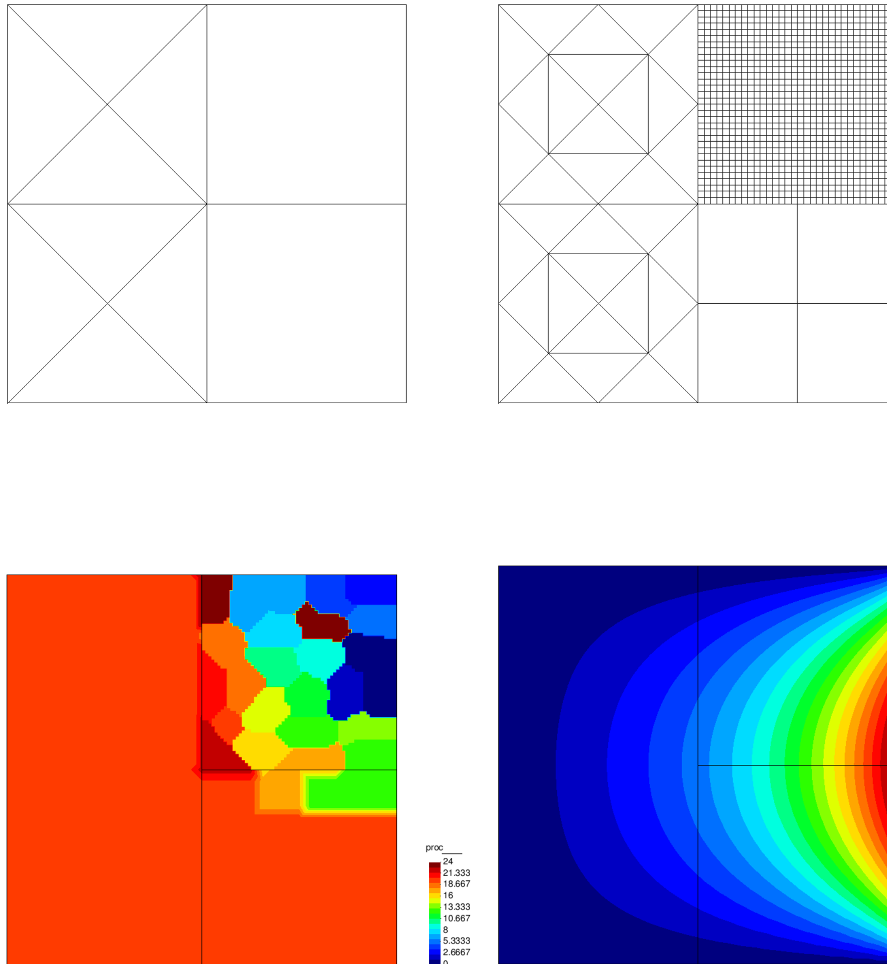

Fig. 10. *Adaptive simulation in a finite element mesh with several types of elements. Top-left: original mesh. Top-right: mesh after some refinement steps. Bottom-left: node distribution across processors. Bottom-right: temperature field at the end of the simulation.*

This numerical example deals with the flow around a cylinder at $\mathrm{Re} = 100$. The computational domain consists of a 16×8 rectangle with a unit-diameter cylinder centered at $(4, 4)$. The horizontal inflow velocity is set to 1 at $x = 0$. Slip boundary conditions are set at $y = 0$ and $y = 8$, and velocity is set to $\mathbf{0}$ at the cylinder surface. The viscosity has been set to $\nu = 0.01$, and the Reynolds number is $\mathrm{Re} = 100$ based on the diameter of the cylinder and the inflow velocity. A third-order backward differences scheme has been used for the time integration with time step size $\delta t = 1 \cdot 10^{-3}$. As an error estimator, the Zienkiewicz–Zhu error estimator [23] for the velocity gradient has been used, which results in a refinement strategy near the boundary layer and in the regions surrounding the vortices behind the cylinder:

$$e_K = \int_K \Pi_\perp \left(\nabla \boldsymbol{u}_h \right) = \int_K \left(\nabla \boldsymbol{u}_h - \Pi_h \left(\nabla \boldsymbol{u}_h \right) \right),$$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

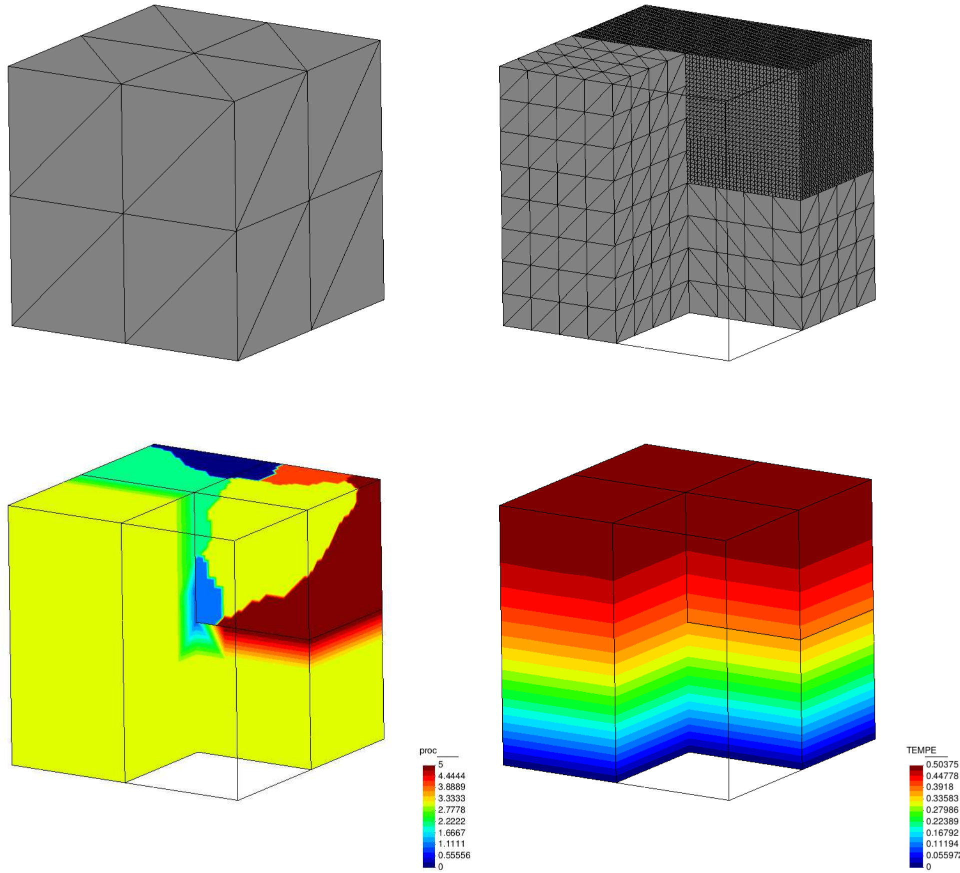

FIG. 11. *Mesh refinement for tetrahedral elements. Top-left: original mesh. Top-right: mesh after some refinement steps. Bottom-left: node distribution across processors. Bottom-right: temperature field at the end of the simulation.*

where K denotes each element of the mesh, Π_h denotes the projection onto the finite element space, and $\Pi_\perp$ denotes the projection onto the space orthogonal to the finite element space. Figure 13 shows the results and mesh evolution obtained for this example. Note that in the refinement process, some of the normal nodes become hanging nodes and vice versa at each step.

6.5. An application to a nonsmooth solution. Here we present a nonsmooth solution example for the steady Stokes problem, which consist of finding $\boldsymbol{u} : \Omega \times (0,T) \longrightarrow \mathbb{R}^d$ and $p : \Omega \times (0,T) \longrightarrow \mathbb{R}$ such that

$$\begin{aligned} -\nu\Delta\boldsymbol{u} + \nabla p &= \boldsymbol{f} \quad \text{in } \Omega, \\ \nabla\cdot\boldsymbol{u} &= 0 \quad \text{in } \Omega, \\ \boldsymbol{u} &= \bar{\boldsymbol{u}} \quad \text{on } \Gamma, \end{aligned}$$

where $\Omega \subset \mathbb{R}^2$ is a two-dimensional bounded domain, ν is the viscosity, and $\boldsymbol{f}$ is a given source term. A divergence free nonsmooth manufactured solution is considered:

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

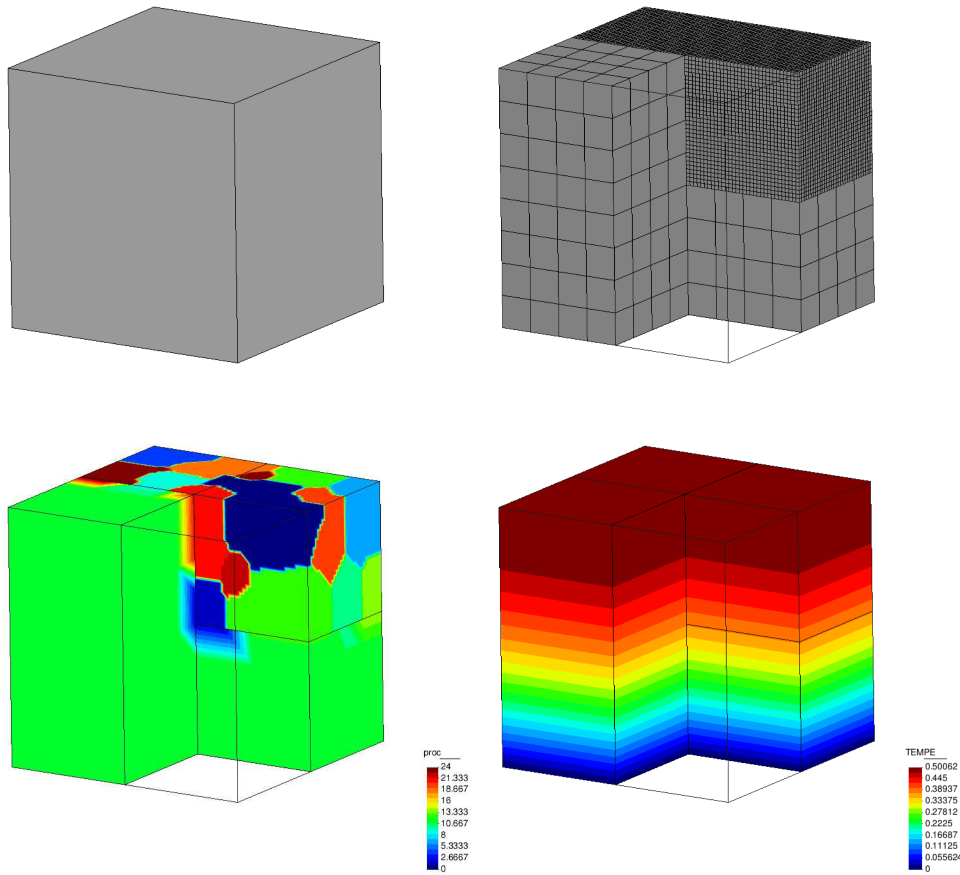

Fig. 12. *Mesh refinement for hexahedral elements. Top-left: original mesh. Top-right: mesh after some refinement steps. Bottom-left: node distribution across processors. Bottom-right: temperature field at the end of the simulation.*

$$
\begin{aligned}
\boldsymbol{u}(r,\phi) &= r^{\alpha}\left[\begin{array}{c} \cos(\phi)\,\psi'(\phi) + (1+\alpha)\sin(\phi)\,\psi(\phi) \\ \sin(\phi)\,\psi'(\phi) - (1+\alpha)\cos(\phi)\,\psi(\phi) \end{array}\right], \\
p(r,\phi) &= -r^{(\alpha-1)}\frac{(1+\alpha)^2\,\psi'(\phi) + \psi'''(\phi)}{1-\alpha},
\end{aligned}
$$

with

$$
\begin{aligned}
\psi(r,\phi) = {} & \frac{\sin((1+\alpha)\,\phi)\cos(\alpha\omega)}{1+\alpha} - \cos((1+\alpha)\,\phi) \\
& + \frac{\sin((\alpha-1)\,\phi)\cos(\alpha\omega)}{1-\alpha} + \cos((\alpha-1)\,\phi)\,.
\end{aligned}
$$

Here ω and α are taken as $\omega = 3\pi/2$ and $\alpha \approx 0.5444837$, which is an approxima-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

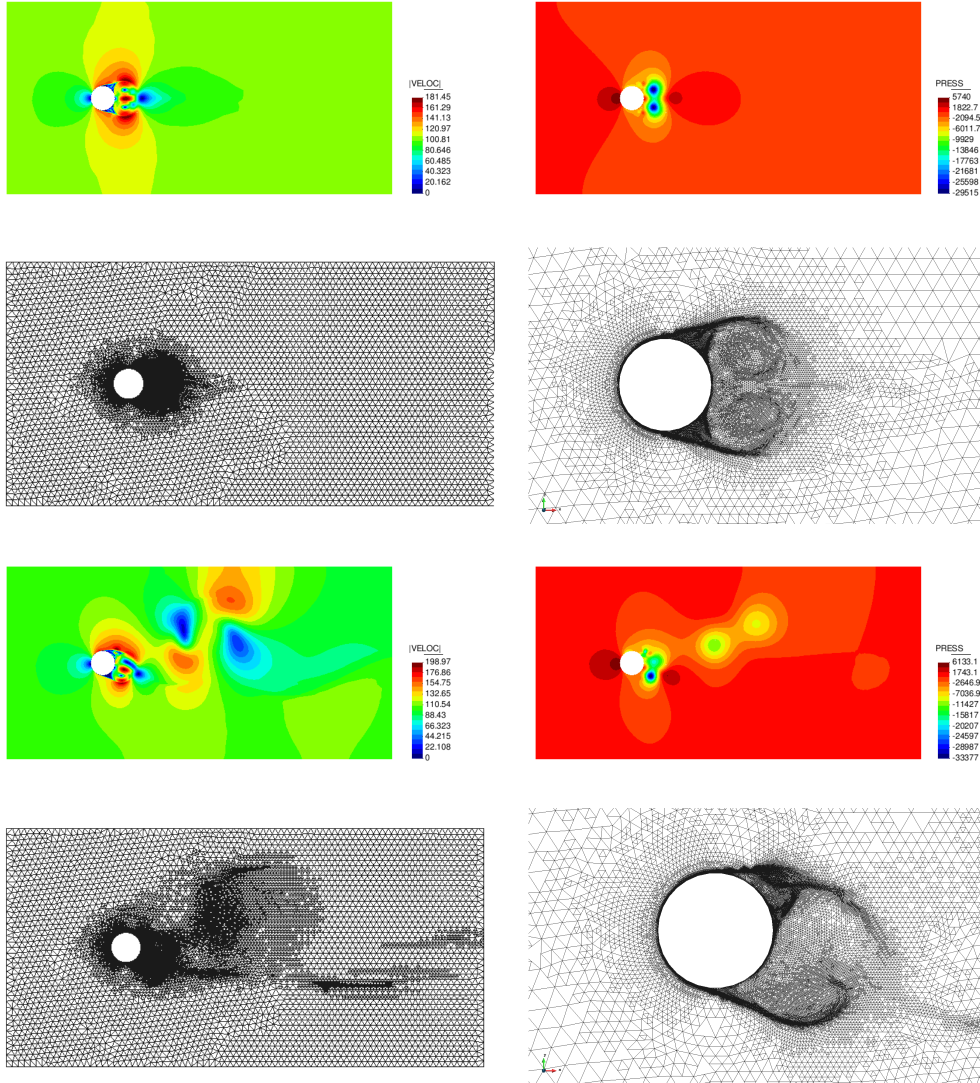

FIG. 13. *Adaptive refinement finite element solution of the flow past a cylinder. Contour results for the velocity and pressure fields, and the refined mesh details, are presented for two separated configurations of the wake oscillation.*

tion to the root of the nonlinear equation

$$\frac{\sin^2(\alpha\omega) - \alpha^2 \sin^2(\omega)}{\alpha^2} = 0.$$

We depart from a six linear element triangular mesh and refine it by using the Zienkiewicz–Zhu error estimator and refinement criteria explained in the previous example. Figure 14 shows the obtained velocity and pressure solutions. The original and the refined mesh after several refinement steps are also displayed. Several refinement levels are developed for the refined mesh. The mesh refinement deals with

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

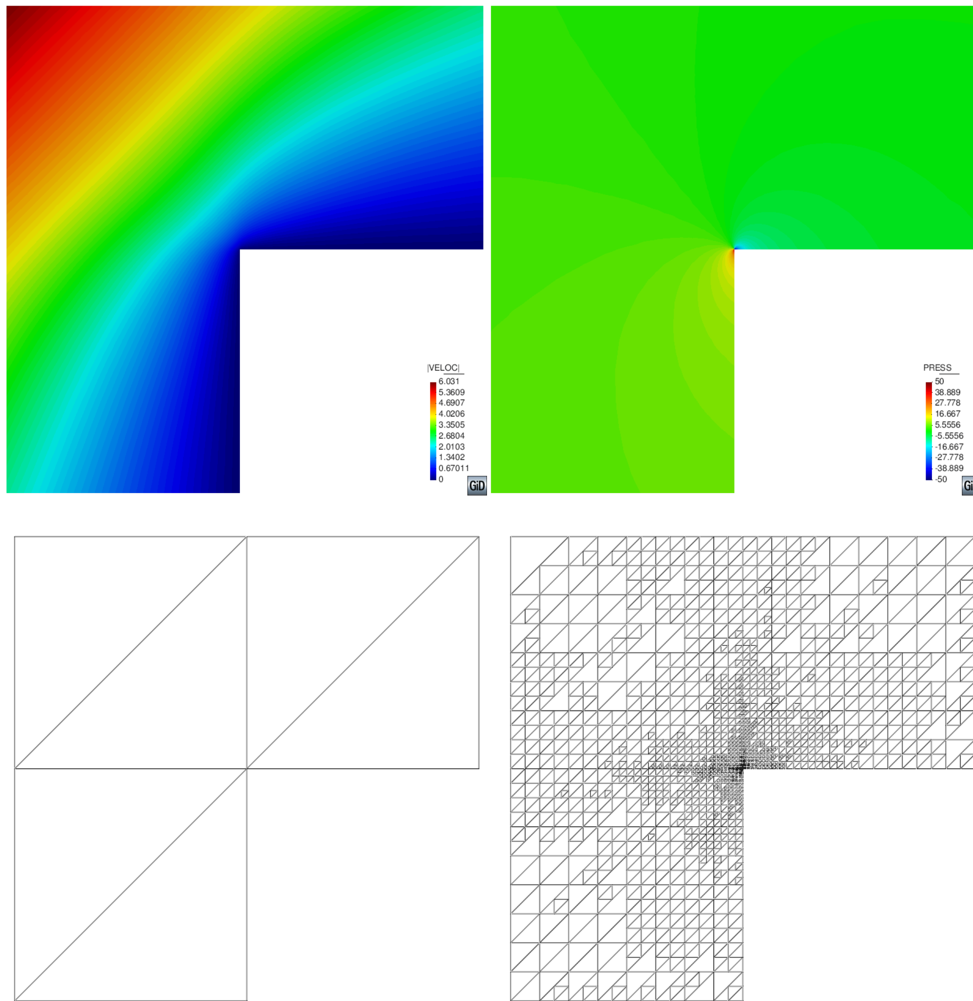

Fig. 14. *Adaptive refinement finite element solution of the Stokes problem, nonsmooth solution. Contour results for the velocity (top-left) and pressure (top-right) fields, and the original (bottom-left) and refined (bottom-right) meshes.*

the singularity appearing in the corner and provides an accurate solution capable of better representing the pressure field close to the singularity.

6.6. An application to free surface flows. This numerical example consists of the simulation of the water flow over the deck of a ship followed by the impact of the resulting wave against an obstacle. This is a well-known benchmark for free surface problems and has been studied experimentally by the Maritime Research Institute Netherlands (MARIN). Experimental data are available in [18]. The setting of the problem is the following. A large tank with an open roof is considered. At one side of the tank, a gate isolates a volume of water which lays at rest at a constant free surface height. On the other side of the tank there is a prismatic obstacle representing a container on the deck of a ship. This obstacle is static, and no fluid-structure interaction effects are considered. The experiment starts when the gate quickly opens

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

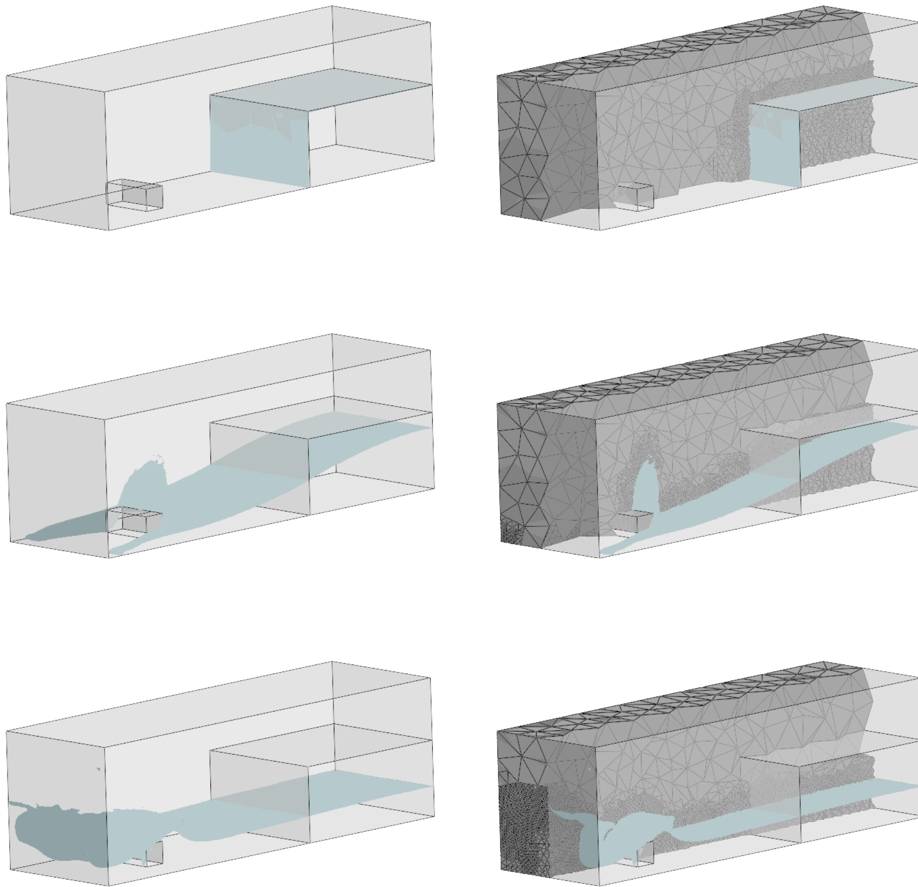

Fig. 15. *Position of the free surface and computational mesh at several instants of the simulation for the green water flow case.*

(the duration of the gate-opening process is considered to be zero) and the resulting water wave impacts against the obstacle and the walls of the tank. The pressure time history at several points on the surface of the obstacle is then measured and compared against experimental results.

An initial mesh composed of 3756 linear tetrahedra elements is used. The time step is set to 0.01 s (seconds) and a total of 6 s is simulated. The finite element mesh is then successively adapted, and the mesh refinement is heavily localized around the free surface, which leads to important savings in the computational effort.

In Figure 15 the position of the free surface at several time instants is shown, together with the mesh at each of these time steps.

6.7. Scalability tests. In this section we test the scalability of the proposed refinement method when a large number of processors is used. We consider two cases. In the first case, we solve an adaptive refinement case where the problem is balanced, and no load rebalancing is required. In the second case, we test the algorithm for a

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

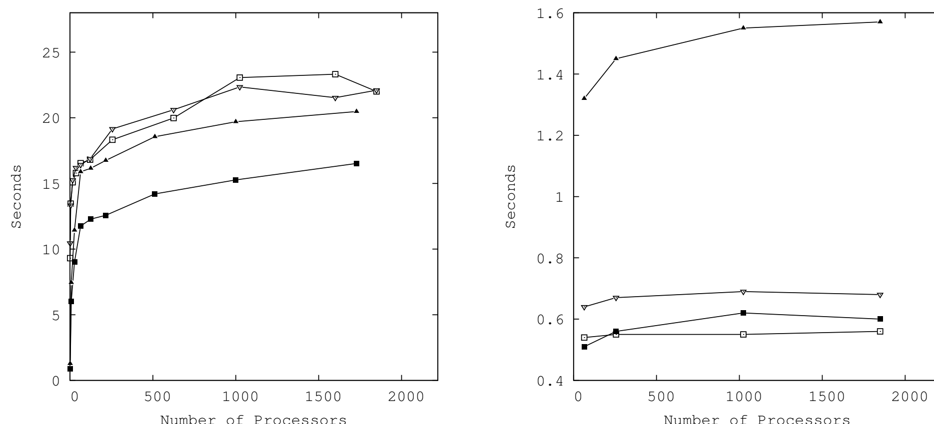

Fig. 16. *Uniform refinement weak scalability results up to* 1849 *processors. The notation for the element type is as follows: Triangles(△), squares (□), tetrahedra(▲), hexahedra(■). Left: in this case we depart from an initial points per processor configuration (*100 *for bidimensional elements and 8 for three-dimensional elements) and perform* 6 *levels of uniform refinement. Thereafter, we include several uniform unrefinement/refinement steps after the last refinement level. This type of run is performed in order to be able to measure the overall refinement performance of several time steps, and it is calculated using the Marenostrum supercomputer. Right: in this case we depart from an initial points per processor configuration (*90,000 *for bidimensional elements and* 270,000 *for three-dimensional elements) and perform a single level of uniform refinement. We calculate this case using the Beskow supercomputer.*

case where load rebalancing is required at almost every step of the refinement process. For each step we refine the elements contained inside the domain, following either a uniform refinement criterion or a refinement criterion which aggressively forces load rebalancing. The weak scalability tests are run up to 1849 processors for bidimensional elements, and 1728 processors for three-dimensional elements. Both scalability cases are limited to 2000 million total elements for the largest mesh created at the last refinement level, since our current implementation uses 4 byte integers for the nodal and elemental counters. The tests presented in this subsection were run on the MareNostrum supercomputer at the Barcelona Supercomputing Center, and on the Beskow supercomputer at KTH, Sweden. The MareNostrum supercomputer is equipped with Intel SandyBridge-EP E5-2670 cores at 2.6 GHz (3056 compute nodes) and 103.5 TB of main memory. The Beskow supercomputer is a Cray XC40 system based on Intel Xeon E5-2698v3 cores at 2.3 GHz (1676 compute nodes) and 104.7 TB of main memory.

The first weak scalability test corresponds to a uniform refinement problem. We depart from a structured uniform mesh with an initial number of elements and points per processor, and successively refine it. For each time step we refine the elements in the entire spatial domain. The CPU runtime invested in the refinement procedures is presented in Figure 16. The objective of this weak scalability test is to measure the communications between processors. Because no communications are needed for load balancing, the measured time is due only to the refinement procedure. The results show an increase in runtime between 1 and 100 processors. After 100 processors a flat tendency is observed. The scaling results are good for both bidimensional and three-dimensional cases, ensuring the correct behavior of the adaptive refinement

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

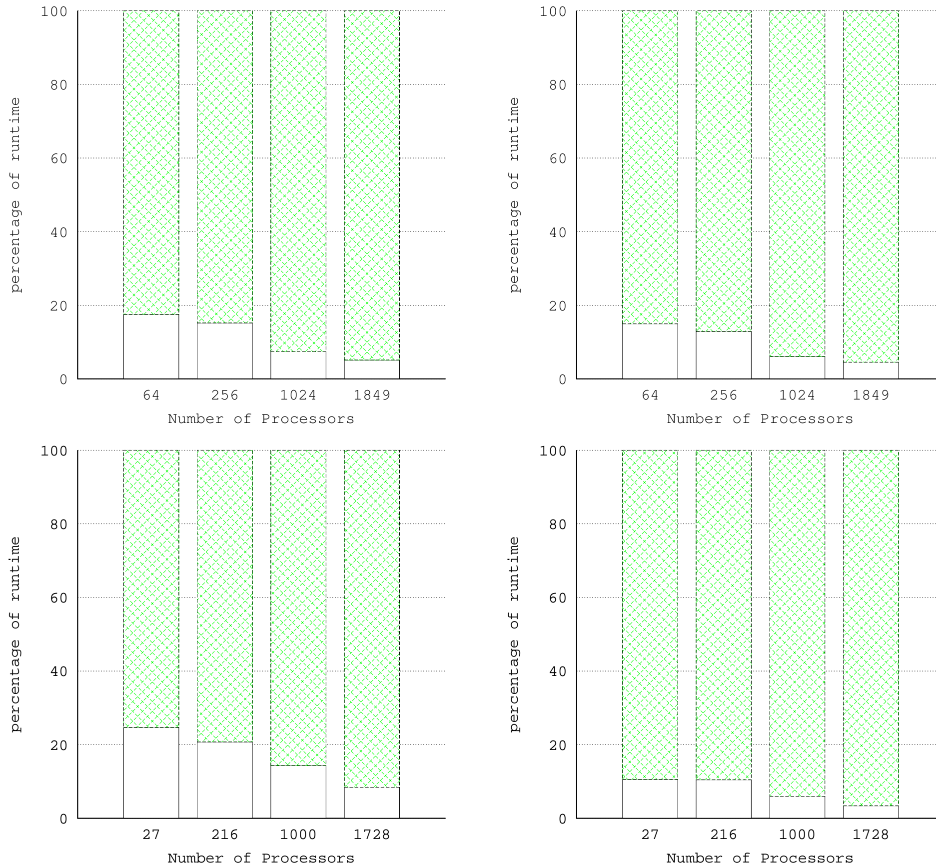

Fig. 17. *Runtime fraction results for the uniform refinement weak scaling tests up to* 1849 *processors on the Beskow supercomputer. Triangles (top-left), squares (top-right), tetrahedra (bottom-left), and hexahedra (bottom-right) element type cases are evaluated. The description of this case is given in Figure* 16*. We perform a single level of uniform refinement and solve the stationary heat transfer problem with the resulting refined mesh. The refinement procedure runtime fraction is plotted with a solid fill, and the linear system runtime fraction is plotted using the pattern fill. We addressed the fraction of time required by a single refinement procedure in comparison with the time spent by the linear solver in order to solve the refined mesh linear system. We compared our results with best runtime from among several linear solvers, which was given by the biconjugate gradient method implemented in the PETSc parallel solver library* [1]*, together with the ML preconditioning package implemented in the TRILINOS library* [10].

algorithm. The runtime fraction of the refinement procedure with respect to the linear system solution is presented in Figure 17 for a single time step. Results show that the runtime is dominated by the linear system solution, and that the runtime fraction of the refinement procedure decreases with the number of processors, ensuring a good behavior in large scale computing.

The second weak scalability test intends to evaluate the overall performance when load rebalancing is necessary. For this case we depart from a structured uniform mesh and refine it successively using the following refinement criteria. For each step we refine only the elements contained inside the domain $\Omega_r = (1-\frac{1}{2^i}, 1) \times (1-\frac{1}{2^i}, 1) \times (0, 1)$, where i denotes the step number in this case. This criterion defines a spatial distribution which partially refines elements of the domain. Figure 18 shows the runtime

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

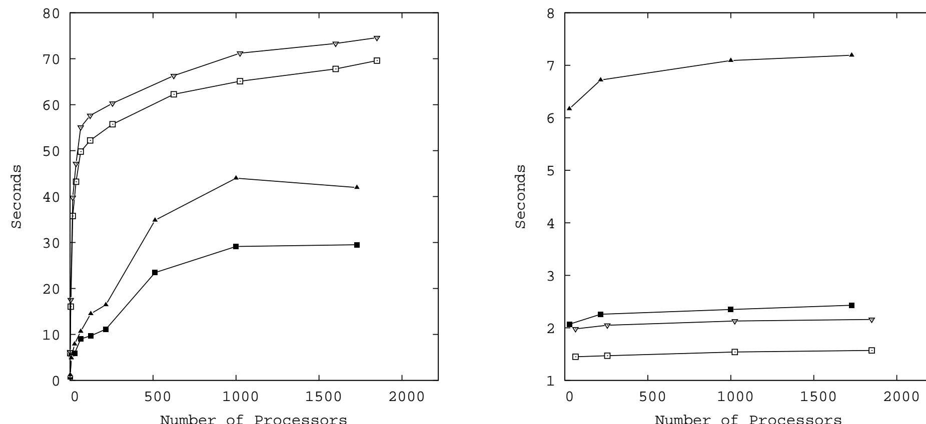

Fig. 18. *Load balancing refinement weak scalability results up to* 1849 *processors. The notation for the element type is as follows: Triangles* (△)*, squares* (□)*, tetrahedra* (▲)*, hexahedra* (■)*. Left: in this case we depart from an initial points per processor configuration (*4900 *for bidimensional elements and* 8 *for three-dimensional elements) and perform* 15 *levels of load rebalancing refinement. This run is calculated using the Marenostrum supercomputer. Right: in this case we depart from an initial points per processor configuration (*90,000 *for bidimensional elements and* 270,000 *for three-dimensional elements) and perform a single level of forced load rebalancing refinement. We calculate this case using the Beskow supercomputer.*

required for the refinement and load balancing procedures. The interprocessor communications in this case are due not only to the refinement step but also to the load rebalancing procedure. An asymptotic tendency is reached for both bidimensional and three-dimensional elements as the number of processors increases. The three-dimensional elements weak scaling results are good up to the range of 1000 processors. The runtime fraction spent by the refinement and load balancing procedures with respect to the linear system solution is presented in Figure 19. In all cases the refinement and load balancing computational cost was small to moderate with respect to the linear system solve. The linear system solve was tested for a one-degree-of-freedom heat transfer problem. In the case of an incompressible Navier–Stokes problem, nonlinearities and a larger number of degrees of freedom per node would increase considerably the cost of solving the linear system, but the refinement procedure cost would remain the same, leading to an even smaller runtime fraction for the refinement step. Therefore, we conclude that the implementation of the proposed algorithm is efficient with respect to the cost of solving linear systems on adaptively refined meshes, and the algorithm is suitable for large scale problems in high-performance computing environments.

7. Conclusions.

In this paper a novel parallel, hierarchical, and load rebalanced algorithm for adaptive mesh refinement (AMR) and coarsening of unstructured bidimensional and three-dimensional meshes has been presented.

The main features of the proposed algorithm are its suitability for nodally based partitions in distributed memory frameworks and its capability of successively refining and unrefining the mesh in an efficient manner in clusters of up to thousands of processors. Several different types of meshes can be dealt with by the algorithm, including triangular, quadrilateral, tetrahedral, and hexahedral elements, and also meshes with several types of elements.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

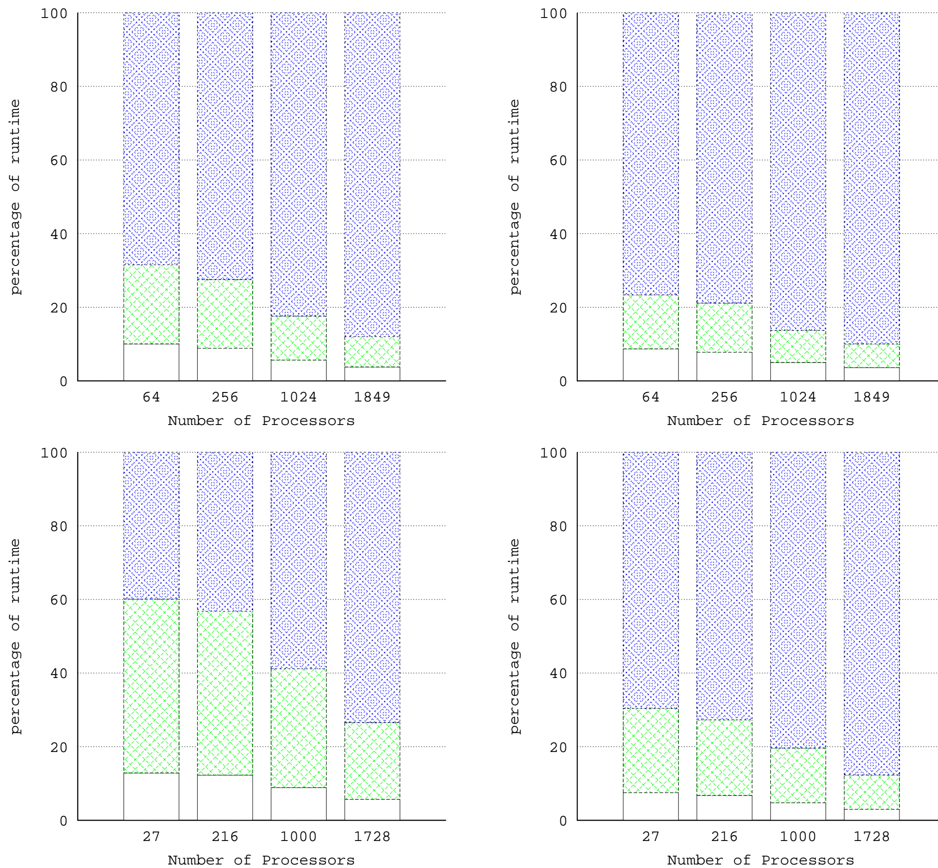

FIG. 19. *Runtime fraction results for the load rebalancing refinement weak scaling tests up to* 1849 *processors on the Beskow supercomputer. Triangles (top-left), squares (top-right), tetrahedra (bottom-left), hexahedra (bottom-right). The full description of this case is given in Figure* 18*. It performs a single level of forced load rebalancing refinement and solves the stationary heat transfer problem with the resulting refined mesh. The linear solver is set to be the biconjugate gradient with the ML preconditioner. The refinement runtime fraction is plotted with a solid fill, the load balancing runtime fraction is plotted with a coarse pattern fill, and the linear system runtime fraction is plotted using the fine pattern fill.*

The memory requirements of the algorithm are reduced at the local level, because only the information corresponding to the part of the mesh in the current processor subdomain needs to be stored locally. The resulting refined meshes are nonconforming with hanging nodes, but no balancing restriction needs to be enforced between adjacent elements, which allows us to have jumps of multiple refinement levels in neighbor elements.

A load rebalancing scheme has been presented which is independent of the rebalancing/renumbering strategy, which can be chosen by the user. Both graph partitioning schemes and space-filling methods (or any other renumbering strategy for load rebalancing) can be used with the proposed algorithm.

Several numerical tests have been presented to illustrate the performance of the proposed algorithm. The first set of tests deals with simulation driven problems such as the Poisson heat transfer problem and the incompressible Navier–Stokes equations

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

in adaptive meshes, for which the expected behavior of the refinement algorithm is obtained. The second set of tests studies the scalability of the algorithm in up to 2000 CPU clusters where good weak scalability results are obtained for meshes of up to 2000 million elements. Also, the runtime fraction for the refinement process is reduced when compared to the runtime for solving linear systems of equations on the generated meshes, which ensures the suitability of the proposed algorithm for large computational physics problems in high-performance computing environments.

The proposed algorithm is packed in the `RefficientLib` Fortran 2003 library for which the user interface has been presented. This allows the easy integration (with no more than 10 calls to the adaptive refinement library) of the proposed algorithm with existing computational physics codes.

The algorithm currently deals with h-refinement; the extension of the algorithm to hp-refinement will be a matter of future work.

Acknowledgments. The simulations were done on the MareNostrum supercomputer at the Barcelona Supercomputing Center of the Centro Nacional de Supercomputación (the Spanish National Supercomputing Center) and on the Beskow supercomputer at the Royal Institute of Technology (KTH), Sweden. We acknowledge PRACE for awarding us access to resource MareNostrum based in Spain at Barcelona. The support of the Red Española de Supercomputación (RES) and the European PRACE network is acknowledged.

REFERENCES

[1] S. Abhyankar, M. F. Adams, S. Balay, J. Brown, L. Dalcin, T. Isaac, M. G. Knepley, D. May, K. Rupp, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc (Portable, Extensible Toolkit for Scientific Computation)*, version 3.7, 2016, http://www.mcs.anl.gov/petsc.

[2] S. Badia and J. Baiges, *Adaptive finite element simulation of incompressible flows by hybrid continuous-discontinuous Galerkin formulations*, SIAM J. Sci. Comput., 35 (2013), pp. A491–A516, https://doi.org/10.1137/120880732.

[3] W. Bangerth, C. Burstedde, T. Heister, and M. Kronbichler, *Algorithms and data structures for massively parallel generic adaptive finite element codes*, ACM Trans. Math. Software, 38 (2011), 14.

[4] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine, *The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring*, Sci. Program., 20 (2012), pp. 129–150.

[5] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. Wilcox, *Extreme-scale AMR*, in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, Piscataway, NJ, 2010, pp. 1–12.

[6] C. Burstedde, L. C. Wilcox, and O. Ghattas, *p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees*, SIAM J. Sci. Comput., 33 (2011), pp. 1103–1133, https://doi.org/10.1137/100791634.

[7] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco, *Dynamic Octree Load Balancing Using Space-Filling Curves*, Technical Report CS-03-01, Department of Computer Science, Williams College, Williamstown, MA, 2003.

[8] B. Cockburn and C.-W. Shu, *The Runge–Kutta discontinuous Galerkin method for conservation laws* V: *Multidimensional systems*, J. Comput. Phys., 141 (1998), pp. 199–224.

[9] R. Codina, *A stabilized finite element method for generalized stationary incompressible flows*, Comput. Methods Appl. Mech. Engrg., 190 (2001), pp. 2681–2706.

[10] M. Heroux, R. Bartlett, V. H. R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams, *An Overview of Trilinos*, Technical Report SAND2003-2927, Sandia National Laboratories, Albuquerque, NM; Livermore, CA, 2003.

[11] T. Isaac, C. Burstedde, and O. Ghattas, *Low-cost parallel algorithms for* 2 : 1 *octree balance*, in Proceedings of the IEEE 26th International Parallel & Distributed Processing Symposium, IEEE Computer Society Press, Piscataway, NJ, 2012, pp. 426–437.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

[12] N. Jansson, J. Hoffman, and J. Jansson, *Framework for massively parallel adaptive finite element computational fluid dynamics on tetrahedral meshes*, SIAM J. Sci. Comput., 34 (2012), pp. C24–C41, https://doi.org/10.1137/100800683.

[13] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput. 20 (1998), pp. 359–392, https://doi.org/10.1137/S1064827595287997.

[14] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey, *LIBMESH: A c++ library for parallel adaptive mesh refinement/coarsening simulations*, Engineering with Computers, 22 (2006), pp. 237–254.

[15] A. Langer, J. Lifflander, P. Miller, K.-C. Pan, L. V. Kale, and P. Ricker, *Scalable algorithms for distributed-memory adaptive mesh refinement*, in Proceedings of the IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE Computer Society Press, Piscataway, NJ, 2012, pp. 100–107.

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

[17] R. S. Sampath and G. Biros, *A parallel geometric multigrid method for finite elements on octree meshes*, SIAM J. Sci. Comput., 32 (2010), pp. 1361–1392, https://doi.org/10.1137/090747774.

[18] *S.E.R.I. Community,* http://spheric-sph.org/index.

[19] H. Sundar, R. S. Sampath, and G. Biros, *Bottom-up construction and* 2:1 *balance refinement of linear octrees in parallel*, SIAM J. Sci. Comput., 30 (2008), pp. 2675–2708, https://doi.org/10.1137/070681727.

[20] T. Tu and D. O'Hallaron, *Balance Refinement of Massive Linear Octrees*, Technical Report CMU-CS-04-129, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2004.

[21] T. Tu, D. R. O'Hallaron, and O. Ghattas, *Scalable parallel octree meshing for terascale applications*, in Proceedings of the ACM/IEEE 2005 Conference on Supercomputing, IEEE Computer Society Press, Piscataway, NJ, 2005, p. 4.

[22] T. Tu, D. R. O'Hallaron, and J. C. López, *Etree: A database-oriented method for generating large octree meshes*, Engineering with Computers, 20 (2004), pp. 117–128.

[23] O. C. Zienkiewicz and J. Z. Zhu, *A simple error estimator and adaptive procedure for practical engineering analysis*, Int. J. Numer. Methods Engrg., 24 (1987), pp. 337–357.

Downloaded 06/16/17 to 5.196.89.225. Redistribution subject to SIAM license or copyright; see http://www.siam.org/journals/ojsa.php

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.