SIAM J. SCI. COMPUT.
Vol. 39, No. 2, pp. C65–C95


# 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).

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-

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

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

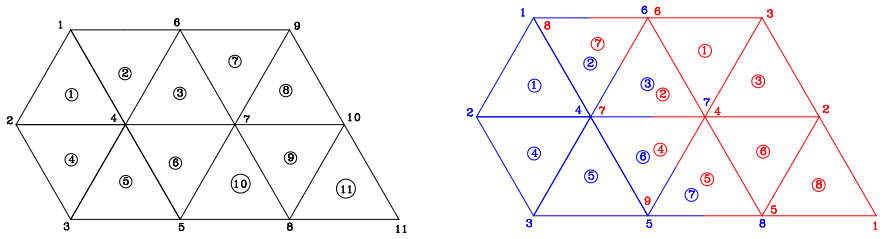

FIG. 1. *Initial mesh information. Left: the global mesh. Points and elements (circled) are numbered globally. Right: the mesh, partitioned into two subdomains. Colors denote the processor to which the information belongs. Points and elements (circled) are numbered locally for each domain. Note that 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.)*

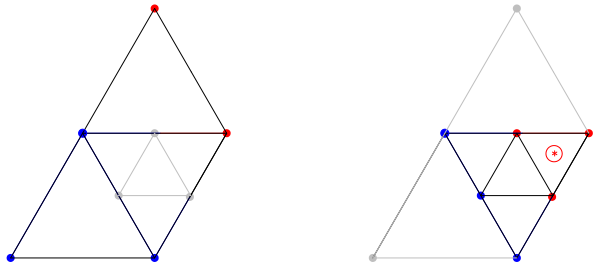

FIG. 2. *Internal hierarchical mesh as seen from one of the processors. In order to illustrate this refinement example, a shared element of the mesh of Figure* 1 *is refined. Left: initial level* 0 *mesh. Right: refined level* 1 *mesh.*

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