Journal of Computational Physics 229 (2010) 3454–3473

Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

Parallel adaptation of general three-dimensional hybrid meshes

Christos Kavouklis *, Yannis Kallinderis

Department of Aerospace Engineering and Engineering Mechanics, The University of Texas at Austin, WRW 303AA, 1 University Station Stop C0600, Austin, TX 78712-1085, USA

ARTICLE INFO

Article history:
Received 3 June 2008
Received in revised form 11 November 2009
Accepted 9 January 2010
Available online 20 January 2010

Keywords:
Parallel computing
Hybrid meshes
Mesh adaptation
Load balancing
Data structures

ABSTRACT

A new parallel dynamic mesh adaptation and load balancing algorithm for general hybrid grids has been developed. The meshes considered in this work are composed of four kinds of elements; tetrahedra, prisms, hexahedra and pyramids, which poses a challenge to parallel mesh adaptation. Additional complexity imposed by the presence of multiple types of elements affects especially data migration, updates of local data structures and interpartition data structures. Efficient partition of hybrid meshes has been accomplished by transforming them to suitable graphs and using serial graph partitioning algorithms. Communication among processors is based on the faces of the interpartition boundary and the termination detection algorithm of Dijkstra is employed to ensure proper flagging of edges for refinement. An inexpensive dynamic load balancing strategy is introduced to redistribute work load among processors after adaptation. In particular, only the initial coarse mesh, with proper weighting, is balanced which yields savings in computation time and relatively simple implementation of mesh quality preservation rules, while facilitating coarsening of refined elements. Special algorithms are employed for (i) data migration and dynamic updates of the local data structures, (ii) determination of the resulting interpartition boundary and (iii) identification of the communication pattern of processors. Several representative applications are included to evaluate the method.

1. Introduction

The Navier–Stokes equations are a standard mathematical representation of viscous flow. Their numerical solution in three-dimensions remains a computationally intensive and challenging task, despite recent advances in computer speed and memory. A widely used strategy to increase accuracy of a Navier–Stokes simulation, while maintaining computing resources to a minimum, is local adaptation of the associated computational grid. Grid adaptation is a twofold technique involving local refinement of the mesh in regions of large solution gradients and coarsening in regions of insignificant solution variation. With the advent of distributed memory architectures and the development of parallel CFD solvers a natural need for parallel adaptation arose. Moreover, adapting the global mesh in a single computer node not only hinders efficiency of the parallel solver but is also impractical for large meshes due to memory limitations, hence the need for parallel adaptation within a distributed memory framework. Another reason for parallelizing the adaptation algorithm is the drop in performance of the overall simulation due to a serial adaptation module. Indeed, as is mentioned in [1], if 10% of the overall simulation code is devoted to serial mesh adaptation and the rest to the parallel solver, then the maximum speed-up we can expect is only 10, regardless of the number of processors we are employing. This fact is easily derived from Amdahl's law:

* Corresponding author. Tel.: +1 650 320 8456.
E-mail addresses: christos.kavouklis@gmail.com (C. Kavouklis), kallind@veltisto.net (Y. Kallinderis).

0021-9991/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2010.01.011

$$S = \left[\frac{p}{N} + (1-p)\right]^{-1}$$

where S is parallel speed-up, N is the number of processors and p is the fraction of code that is parallelizable.

The present work focuses on parallel adaptive general hybrid meshes. General hybrid meshes exhibit a structured nature near viscous walls where hexahedra and prisms are used. The rest of the computational domain is covered with tetrahedra, while pyramids are employed to match the structured and unstructured portions of the mesh. A typical hybrid mesh for a cylinder geometry is presented in Fig. 1. The mesh is intentionally very fine in the rear region to accurately resolve the flow wake. The generation of meshes used in this work and a survey of hybrid unstructured mesh techniques can be found in [2,3].

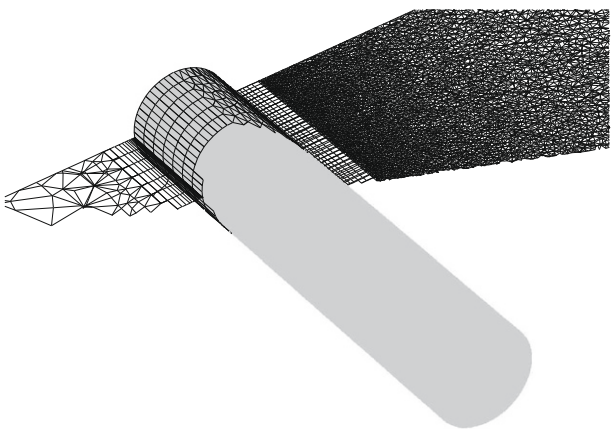
Several issues need to be addressed within the context of a parallel adaptation strategy; mesh partitioning, parallel refinement/coarsening and redistribution of load. The design of a suitable data structure to handle parallel communication is also essential. In the following subsections, previous research efforts on these topics are discussed.

1.1. Partitioning of 3D meshes

Mesh partitioning is a problem with a dual objective; at first, a good balance of load must be realized (i.e. partitions should have roughly the same number of elements), and secondly, interprocessor communication should be minimal. The latter goal is achieved by minimizing the size of the interprocessor boundary.

The spectral bisection approach reduces partitioning of a mesh to a graph partitioning problem, since a mesh can be interpreted as a graph [4]. In particular, the following problem is solved for partitioning a graph into two partitions: minimize $\frac{1}{4}\mathbf{x}^T L\mathbf{x}$, where $x_1, \ldots, x_N \in \{-1, +1\}, \sum_{i=1}^{N} x_i = 0$, N is the number of grid elements, L_{ii} is the number of vertices connected to vertex i, $L_{ij} = -1$ if vertices i,j constitute an edge and $L_{ij} = 0$ otherwise. This constrained optimization problem is equivalent to computing the second eigenvalue (and its corresponding eigenvector) of matrix L [5,6]. Spectral bisection can then be recursively applied to partition a graph into an arbitrary number of partitions that is a power of 2. Computing the second eigenvalue of L and its eigenvector is expensive, however, the computation time may be significantly reduced using a conjugate gradient method approach [7]. A very successful strategy has been studied in [8–10], where spectral bisection is implemented within a multilevel framework to accelerate the algorithm. In particular, the original graph is successively coarsened down to a much smaller graph and then spectral bisection is employed on the coarsest graph. Finally, the resulting partition is sequentially projected to the initial graph. The advantage of these strategies, in contrast with conventional geometric partitioning schemes, is minimization of the size of the *edge-cut* of partitions, which in return, results in a smaller communication cost.

A geometry-based partitioning method is recursive coordinate bisection. In this instance, mesh elements are assigned to processors by recursively dissecting the computational domain using cutting planes across either the coordinate axes (Orthogonal Recursive Bisection, ORB) or the principal axes of the mesh (Inertial Recursive Bisection, IRB) [11]. The method is fast and yields contiguous partitions. However, compared with spectral bisection it results in larger interpartition boundaries. The size of the interpartition boundary may be reduced by applying recursive coordinate bisection not on the hybrid mesh itself but on the leaves (*octants*) of the mesh octree [12,13]. A mesh can also be partitioned by using a depth first



traversal of the octree [14,15]. This method preserves locality, that is, elements within a partition are spatially adjacent, however it does not offer any control on the size of the interpartition boundary.

The RCM (Reverse Cuthill McKee) algorithm [16] may also be utilized for mesh partitioning. The algorithm renumbers the vertices of the graph associated with the hybrid mesh [4] so that the bandwidth of its adjacency matrix is reduced. By breaking the new ordered list of elements in p segments, a partitioning of the mesh with p partitions that are contiguous is obtained.

Another partitioning method, based on the space filling curve (*Hilbert curve*), was investigated in [17] for the case of a uniform mesh in a square and in [18,19] for octrees of unstructured meshes. The space filling curve passes through the centroids of all mesh elements (or leaf octants), hence dividing it into segments yields a partitioning of the mesh. A property of the *Hilbert* curve is that any pair of consecutive elements (or leaf octants) in the associated ordered list are face neighbours, which results in contiguous partitions. The method is very fast. However, it lacks control of the interboundary size.

For all of the aforementioned approaches, given two neighbour partitions, the Kernighan–Lin algorithm [20] may be used to further decrease the length of the interprocessor boundary by mutual exchange of graph vertices. Similarly, in [13–15] local exchange of elements is employed to eliminate protruding interpartition boundaries and further reduce their length and hence the communication cost.

In the present work, the schemes of [8–10] are employed for general hybrid mesh partitioning, since they provide a minimal interpartition boundary size at an execution time comparable to the fast geometry-based partitioning methods.

1.2. Dynamic load balancing on 3D meshes

Local refinement or coarsening of the mesh produces an imbalance of load shared by processors. As a result, the mesh must be equidistributed among processors so that performance of the numerical solver does not deteriorate.

The simplest dynamic load balancing approach is based on merging of local meshes and having a *master* processor repartitioning the global mesh [21–23]. However, overloading one processor with partitioning may result in deterioration of performance. Another deficiency of this strategy is that new partitions may be significantly different from the previous ones and large data sets may need to be migrated, depending on the partitioning method.

In [24] the octree partitioning technique of [13] is used to dynamically balance an adapted hybrid mesh. Specifically, each processor holds a copy of the initial global octree corresponding to the coarse mesh. After refinement, the weights of the leaf octants are adjusted based on the number of elements they contain and then each processor partitions the weighted octree as in [13] using recursive coordinate bisection. The method is incremental, that is new and old partitions do not vary appreciably and hence data migration is not costly. However, for large meshes and a large number of processors this method results in some loss of performance, since it has a constant speed-up of 1 [14].

The octree partitioning scheme of [14,15], described in the previous subsection, may also be used for dynamic load balancing. In [15,18] each processor stores a portion of the octree and during the load balancing phase, a local depth first traversal is employed to assign sub-octrees to partitions.

In [14,25] a parallel inertial recursive bisection algorithm is introduced. Since IRB requires the computation of the median value of a set of doubles (either the x or y or z inertial coordinates) its parallel version can be reduced to efficient parallel sorting. The method will in general give very different partitions from the original ones, due to the sensitivity of inertia axes to mesh alteration (see, for instance, [15] for a relative example), and will hence induce a high data migration cost. However, it yields better quality partitions compared to the octree dynamic load balancing of [15,18].

Authors of [14,25] also refer to a dynamic load balancing strategy based on iterative tree balancing. In this context, each processor requests load from its neighbour with the highest load difference and the communication graph of processors is separated into a forest of trees. Then, the trees are balanced and the procedure is repeated until load is equidistributed. The method is non-deterministic and at each iteration data has to be migrated. Further, it gives lower quality partitions compared to the octree load balancing approach of [15,18] and parallel IRB.

A *divide and conquer* load balancing method is presented in [11]. The processors initially form two groups and the group with the highest load migrates elements to the other one. The transfer of elements involves only processors that are face neighbours. The procedure is then recursively applied to each one of the equally loaded processor groups. The algorithm is deterministic and converges in $log(p)$ steps, where p is the number of processors and it is a power of 2. However, it lacks control of the interpartition boundary size and data must be migrated at each step.

In [26–28] the load balancing strategy that refinement trees of coarse elements (i.e., elements resulting from multiple refinement of initial mesh elements) reside on the same processor is employed. This requirement, which has been adopted in the present work for general hybrid meshes, results in straightforward application of mesh quality preservation rules and facilitates mesh coarsening. In [26,27] grid load balancing is based on repartitioning the weighted dual graph associated with the grid using the *Parmetis* library [29,30]. The authors note that assigning new partition i to processor i is not necessarily the best strategy to reduce data transfer. Instead, a special algorithm is employed to optimally assign partitions to processors, so that the data migration cost is minimized.

In the present work, the *Zoltan* library [19] has been utilized for dynamic load balancing, using the *Parmetis* graph repartitioning algorithms [31]. *Parmetis* provides fast partitioning time and partitions of superior quality combined with a reduced data migration cost, due to optimal distribution of partitions to processors. A survey of dynamic load balancing approaches can be found in [32].

1.3. Parallel data structures

The data structures pertaining to the interpartition boundary among processors are a crucial feature of any parallel adaptation algorithm, since communication of tagged edges for refinement is based on them. Furthermore, special care needs to be taken for the data structures that support dynamic load balancing.

In [13,24,27] dynamic lists of interpartition edges and nodes are used, while in [33] an additional list of interpartition faces is maintained. Similarly, in references [14,25,34] the interpartition boundary data structure consists of faces, edges and nodes. Interpartition entities shared by any two processors form doubly linked lists for easy insertions and deletions. To facilitate communication, each shared mesh entity stores the *ids* of processors that share it, as well as the local *ids* of its copies in these processors. Additionally, in [13,24], the octree associated with the coarse hybrid mesh is stored in each processor and is used for dynamic load balancing, as mentioned in the previous subsection.

In [14,15,25] a distributed octree is employed for dynamic load balancing. Each octant that is not present in the local processor is replaced by a structure that contains its owner processor and address. Similarly, non-present parents of octants are replaced by auxiliary structures that contain their owner processors and remote locations. In general, implementing a distributed octree, as opposed to [24], is a complex task, however its use is imperative for very large meshes.

In [35] mesh nodes are partitioned across processors. Lists of shared edges are utilized for communicating refinement marks. The authors do not use the usual tree data structure to handle refinement/derefinement of elements, instead they are storing them in a hash table. An additional table of element edges is constructed and, by means of it, children of a given element can be accessed.

In [21] a tree data structure of interpartition faces is employed for the case of all tetrahedra meshes. Processors communicate only triangular faces of the interpartition boundary and binary patterns corresponding to their marked edges. A consistent ordering of nodes of interpartition faces is essential for this method. It is also required in the parallel refinement scheme of [36] to ensure compatible refinement patterns of interpartition faces.

In the present work, a new simple parallel data structure has been designed. The only interpartition boundary entities required for parallel communication are the interprocessor faces. No additional data structure is required to accommodate them, since they are stored in the hash table of boundary faces. Moreover, a special ordering of their nodes is not needed. A small hash table that transforms global *ids* of nodes to local *ids* is also defined for parallel communication. It is dynamic, hence more appropriate for parallel adaptation, compared to the conventional static translation tables. Finally, the weighted dual graph corresponding to the adapted hybrid mesh is stored in each processor using the CSR format [37] and is utilized for dynamic load balancing.

1.4. Parallel refinement and coarsening

In a parallel refinement/coarsening procedure, each processor typically refines/coarsens its portion of the mesh. Communication is necessary to adjust interpartition boundary entities and ensure consistency of local meshes.

In reference [35], as well as in the present work, refinement is based on exchange of edge refinement flags. Element refinement involves division of the oldest edge or of the longest one, in case of a tie.

In [38–40] local meshes are first refined excluding elements close to the interpartition boundary. Subsequently, after an element migration step, the interpartition boundaries are shifted. Finally, the remaining elements adjacent to old interpartition boundaries are refined. Despite employment of this cell migration step a significant merit of the method is the absence of communication for altering the interpartition boundary.

A *synchronous* parallel tetrahedral grid adaptation scheme is presented in [41]. The associated serial algorithm is similar to the work in [42], that is, first tetrahedra flagged for refinement are isotropically refined and then hanging nodes are eliminated by employing a closure procedure. The parallel version of the algorithm starts with isotropic refinement of marked local elements. In the sequel, a parallel *while* loop is executed to obtain a conformal mesh. At each iteration, elements not belonging to the permitted division types are refined isotropically and, further, edge marks of interpartition boundary edges are communicated.

In [27] parallel adaptation of tetrahedral grids is based on a *synchronous* iterative edge marking algorithm. In serial, mesh edges are flagged for refinement until all elements obtain refinement patterns that result in a conformal mesh, before refinement is invoked. In the parallel extension of the algorithm, processors first flag their local meshes and upon local convergence they communicate interboundary edge refinement marks. The procedure is repeated if any inconsistency is detected on the interpartition boundaries. Finally, each processor proceeds with refinement of its local mesh.

In [34] children of a refined element may reside on different processors, however they need to be moved in the same processor before invoking the coarsening procedure. Parallel refinement presents no complications since hanging nodes are allowed and therefore each processor can refine its elements independently. It should be noted that *synchronous* approaches do not offer optimal usage of processors.

In [36] all possible division types of a tetrahedron have been derived (42 in total). As a result, refinement does not propagate to neighbours of refined elements (and hence also to neighbour processors in a parallel setting). This property entails a fairly simple parallel refinement algorithm that does not require any communication among processors. In particular, processors first refine their interpartition faces and then proceed with refining their local meshes, independently, without further communication.

In [36,25,39] coarsening is achieved by edge collapsing. In this case, edges flagged for coarsening are deleted and their vertices are collapsed, while elements that share them are removed. In [15] coarsening involves deletion of all elements included within the ball of a node (polyhedron composed of all elements that share the node) and further regriding with fewer elements. Both approaches require communication to identify all elements that share an interpartition edge or node before proceeding with coarsening (except from study [39], where such communication is not required as mentioned before).

In the present work, the hierarchical refinement/coarsening adaptation scheme of [43,46] is parallelized. The parallel refinement algorithm involves the exchange of interprocessor edges' refinement tags. It makes use of Dijkstra's ring algorithm for distributed termination detection [44,45] and is completely *asynchronous*. Also, it is required that descendants of a refined element lie on the same processor. As a result, mesh coarsening is automatic and minimal communication is needed to delete interpartition nodes after derefinement. It is also noted that in references [11,13] only one level of parallel refinement is considered, in contrast with the present work, and specifically in [13] the issue of coarsening is not addressed.

The paper is structured as follows. In Section 2, a few details regarding the serial adapter discussed in [43,46] are agglomerated. Issues pertaining to element refinement types, adaptation rules and data structures are briefly presented. In Section 3, the additional data structures that are needed for parallel communication are discussed. In Section 4, the actual parallel adaptation algorithm is presented. Flow feature detection in parallel is discussed and the application of Dijkstra's algorithm to the communication of interpartition edges' refinement marks is shown. Section 5 deals with a load balancing approach based on the *Zoltan* library [19]. The algorithms related to data migration and local data structures' alteration after mesh repartitioning are also given. In Section 6, an algorithm for the determination of the interpartition boundary after load balancing is described along with a simple procedure that determines the face neighbours of a processor. The paper concludes with Section 7, where several applications are presented. In particular, global refinement of a cylinder mesh, emulation of a shock wave in a channel and subsonic flow about a NACA 0012 wing are considered.

2. The serial mesh adaptation method

Serial refinement and coarsening of general hybrid meshes has been the focus of recent previous work [43,46]. In this section the main aspects of this work are briefly described. The computational algorithm involves the flagging of edges for refinement or coarsening according to some error estimator. Then, the mesh is first coarsened and the refinement procedure is invoked. The latter involves continuous application of the element flagging rules (in a *while* loop) until no further edges are marked for refinement. Finally, the mesh is refined, the solution is interpolated to the newly created nodes and it is passed to the solver module for the next simulation cycle. The most intensive part of the algorithm, in terms of CPU time, is the *while* loop of the application of the element flagging rules.

2.1. Element division types

Division types for tetrahedra, prisms, hexahedra and pyramids have been defined and implemented using special templates. For tetrahedra the usual 1:2, 1:4 and 1:8 divisions are employed, augmented with refinement types compatible to those of pyramids, while directional refinement of prisms and hexahedra is utilized to preserve the structured nature of the mesh along the boundary layer regions. For pyramids both isotropic and anisotropic refinements are permitted to complement the division types of the rest types of elements. Division types are compatible with each other in order to avoid the presence of hanging nodes in the adapted mesh. Representative element division types are depicted in Fig. 2.

For each division type, given the resulting children, the parent element can be reconstructed when coarsening the grid. In particular, children elements and their vertices are carefully numbered, in order for the coarsening algorithm to identify the vertices of parent elements and re-create them [43,46].

2.2. Element flagging rules

Certain rules are applied to obtain a valid mesh without hanging nodes after adaptation and to retain its quality. At first, the faces of elements are visited and their edges are appropriately flagged for refinement. Specifically, if two edges of a triangular face are flagged for refinement, then the third edge is also flagged, and, if all edges of a quadrilateral face are flagged for refinement, then its centroid is introduced.

To preserve quality of the initial mesh, special rules are enforced. In particular, it is required that the maximum adaptation level difference of elements sharing a node is no more than one. This may be required by node based finite volume solvers to ensure that the centroid of node dual volumes is closed to their associated nodes. In addition, no children of an anisotropically refined element may be further refined. Instead, the parent element is reinstated and further isotropically refined (isotropic division rule). The associated algorithm makes use of the coarsening scheme discussed in [43,46]. This rule is essential for avoiding skewed elements which in turn affect the quality of node dual volumes.

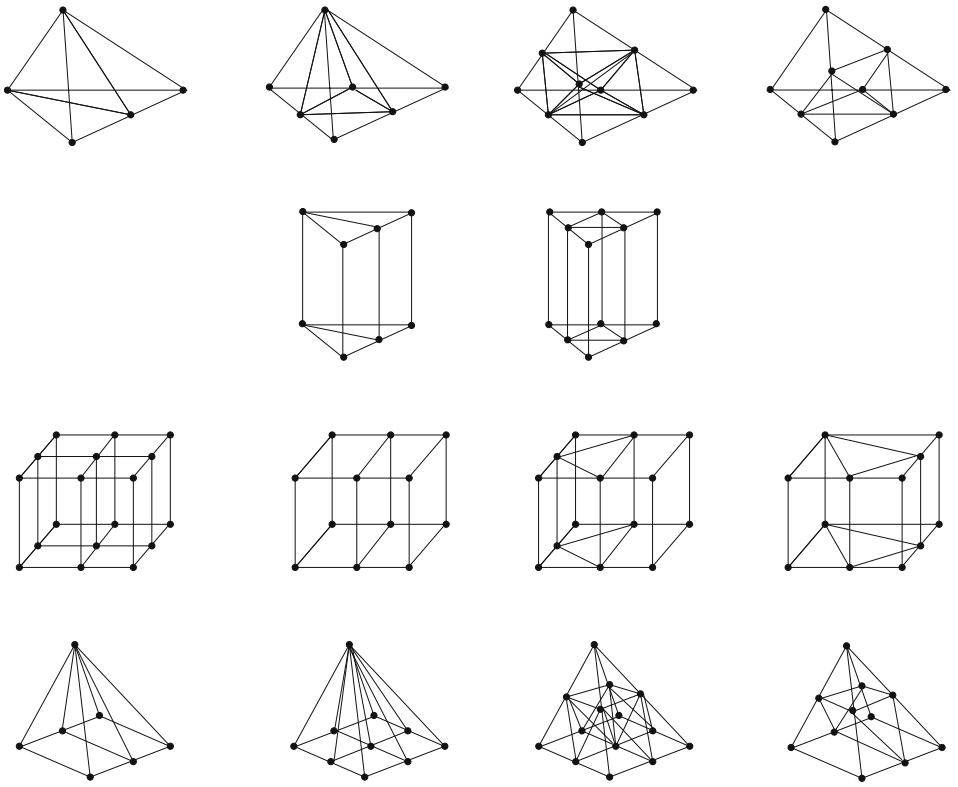


Fig. 2. Representative division types of tetrahedra, prisms, hexahedra and pyramids [43,46]. It is emphasized that refinement of tetrahedra, hexahedra and pyramids may result in children of mixed type.

2.3. Serial data structures

The fundamental entities for representing a hybrid mesh are the *nodes* which correspond to its vertices. The other entities namely *tetrahedra*, *prisms*, *hexahedra*, *pyramids*, *edges* and *boundary faces* are defined by pointers to their vertices. Another entity, the *center*, is defined to account for the centroids of refined quadrilateral faces.

The data structures of the serial adapter are dynamic in nature and allow for multiple levels of refinement/coarsening. To store mesh elements and *nodes* linked lists are used and *edges*, *boundary faces* and *centers* are stored in hash tables. The adjacencies of mesh entities are depicted in Fig. 3. It is noted that parent elements are not stored in the data structure, rather special auxiliary entities that contain the mesh hierarchical information are stored in the element lists. In particular, given a set of sibling elements and the associated auxiliary entity, the coarsening algorithm of [43,46] can reinstate the corresponding parent element in the data structure.

3. Parallel data structures

This section addresses the additional data structures that are defined for the parallel adapter. A dual graph [4] that represents a hybrid mesh is considered. Its vertices are the mesh elements and its edges are the pairs of face adjacent mesh

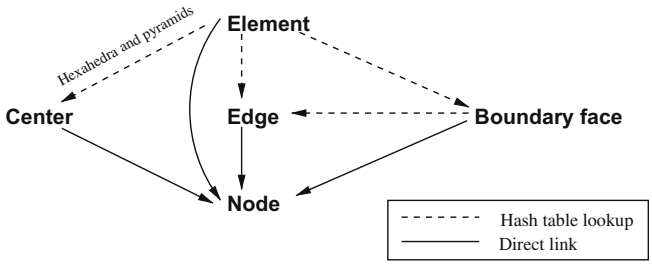


Fig. 3. Relationships among mesh entities. Dotted lines indicate access by table lookup and solid lines indicate direct links.

elements. The dual graph is required for both initial partitioning of the mesh and load balancing of the resulting adapted meshes. By partitioning it, partitions of the associated hybrid mesh are readily obtained. In the present work the Metis [8] and Chaco [47] graph partitioners have been employed. Interpartition boundary data are managed via table lookup and an additional hash table for transforming global *ids* of interpartition nodes to local *ids* is defined.

3.1. Data structure for the dual graph

The dual graph is implemented using the CSR (compressed storage) format. This is composed of four arrays and is not expensive in terms of memory. It is noted that each processor holds the whole dual graph of the original mesh in CSR format and not only that of its own partition.

An important consideration is to enforce the rule that siblings resulting from the refinement of an element are present within the same processor. This is necessary for the adaptation algorithm when applying the isotropic refinement rule or when coarsening a group of sibling elements. If this condition is not satisfied, expensive and complicated data migration is required. An implication of this rule is that after several adaptations the whole refinement tree (i.e., all descendants) of a coarse mesh element resides on the same processor. Additionally, when the mesh is load balanced, only coarse mesh elements and refinement trees may be migrated. Similar ideas concerning the use of the dual graph have been employed before [21,22,26,27]. However, the present work is one of the first to apply them for general hybrid meshes.

3.2. Data structure for the interpartition boundary

Exchange of information between neighbour processors is based on their common interpartition boundary edges. In contrast with other approaches that utilize lists or static arrays of interpartition edges, faces and nodes, a new simple data structure for interpartition boundary entities is proposed. In particular, only faces of the interpartition boundary are employed and interpartition edges are accessed through them by table lookup. The interpartition faces in a given processor are stored in the hash table of boundary faces and are treated as boundary faces by the refinement and coarsening routines. In addition, they are updated dynamically during adaptation.

3.3. Global to local id transform

Most parallel solvers use special local numbering of the elements involved in the numerical stencil to take advantage of *Cache* memory. The adapter may work with only global numbers of nodes, but for compatibility with the solver and efficiency of the adapter's hash functions, local numbering of nodes is also introduced. Hence the adaptation module is acting on local numbers of nodes. For communication among processors global *ids* of nodes are utilized, however an issue arises when a processor receives the global *id* of an interpartition node from a neighbouring processor. Then the copy of the node in the local data structure of the receiver processor cannot be identified. To resolve this problem, a small hash table that converts global *ids* of interpartition nodes to their corresponding local *ids* is defined. With this new approach the usual constant size translation tables are avoided.

4. Parallel mesh adaptation

This section is concerned with the parallel adaptation algorithm. Special care needs to be taken for identifying edges to be refined or coarsened and for communicating refinement flags of interpartition boundary edges. The first phase of the algorithm guarantees that mesh edges are properly flagged for refinement so that no hanging nodes appear after adaptation. The main issues here are the communication among processors of the refinement flags of edges lying on their interpartition boundary and the detection of termination of the edge flagging procedure for all processors. To ensure that all processors have stopped flagging their edges and hence it is safe to proceed with refinement, Dijkstra's algorithm for distributed termination detection [44,45] is employed. On the second phase, each processor is independently refining its portion of the mesh.

4.1. Parallel flow feature detection

Flow feature detection is based on undivided and divided differences of flow quantities along mesh edges, as has been demonstrated in [48,49]. In particular, if ϕ is a flow quantity of interest, then the mean values and standard deviations of undivided and divided differences of ϕ (μ_1, σ_1 and μ_2, σ_2, respectively) are computed. Then, edges with divided and undivided differences above the threshold values $\mu_1 + \alpha\sigma_1$ and $\mu_2 + \alpha\sigma_2$ are refined. Here, α is the adaptation threshold coefficient, which is an empirical parameter. Similarly, edges with divided and undivided differences below the threshold values $\mu_1 - \beta\sigma_1$ and $\mu_2 - \beta\sigma_2$ are flagged for coarsening, where β is another empirical parameter.

The computation of these statistical quantities in parallel is more involved because of interpartition edges that are shared by multiple processors and the necessary communication required. Here, a new parallel feature detection algorithm based on the aforementioned works is presented. To facilitate the analysis, the existence of a *master* processor that collects

information about edges and coordinates the computation, is assumed. The procedure is composed of the following four steps:

(a) Each processor identifies and counts its interior and interpartition boundary edges. Then, the sums of differences of ϕ along interior edges are computed. A message that contains these sums and the interpartition edges is sent to the *master* processor.
(b) The *master* processor uses table lookup to compute the sums of differences along interpartition edges. It then scatters the mean values to all slave processors.
(c) The slave processors compute sums of the form $\sum(D_i\phi - \mu)^2$ over their interior edges and send them to the *master* processor. Here, $D_i\phi$ is a difference of ϕ along edge i.
(d) Finally, the *master* computes similar sums over the interpartition edges and the standard deviations σ_1, σ_2 are evaluated.

Given the mean values and standard deviations of quantity ϕ, the slave processors identify edges that must be refined or coarsened and start the adaptation process. It is emphasized that this is the only place in the present parallel adaptive method where a *master* processor is utilized.

4.2. The need for global termination detection

An important aspect of the proposed parallel mesh adaptation algorithm is communication among processors while local edges are being flagged for refinement. As was seen in Section 2, each processor is executing a *while* loop, that involves application of flagging rules for edges, to obtain a local conformal mesh, in other words a mesh without hanging nodes. Processors must communicate the refinement marks of their interpartition edges to ensure the absence of possible hanging nodes on the interpartition boundary. If at least one interpartition edge of processor: P_i becomes flagged by a neighbour processor P_j then the edge flagging iteration must be repeated for P_i. Even if a processor has completed flagging its edges it is not safe to proceed with refinement of its elements due to non-delivered messages from its neighbours that may incur hanging nodes on the interpartition boundary. Before invoking the mesh refinement scheme it is mandatory to ensure that all processors have finished flagging their edges and that no messages are pending. To detect the global termination of the processor system, Dijkstra's algorithm for distributed termination detection is employed.

In the following, a brief description of Dijkstra's algorithm is presented. Initially, all processors are active. Processors are executing tasks asynchronously and it is assumed that if a processor is idle it cannot be activated spontaneously, but only after receiving a reactivation message from another processor. The goal is to detect whether all processors are terminated. To achieve this state of global termination, two conditions need to be satisfied. First, all processors must be idle, and secondly, no further messages should be in transit. The latter condition is required since an already terminated processor can be reactivated by a pending message.

In the following, it is assumed that processors form a closed logical ring $P_0, P_1, \ldots, P_{n-1}$ where n is the number of processors. In other words, processor P_i follows P_{i-1} and precedes P_{i+1}. Further, processor P_{n-1} precedes processor P_0. It is also assumed that any processor P_i can communicate a token to its following processor P_{i+1}. The token is colored white or black. Moreover, it is augmented with the difference of the total number of sent messages minus the number of received messages, denoted by *Nmsg*. In Fig. 4 the two independent types of communication paths within a system of eight processors are depicted. As is seen, messages that contain information about the global task are communicated, as well as messages that contain the token. Dijkstra's algorithm comprises the following set of rules:

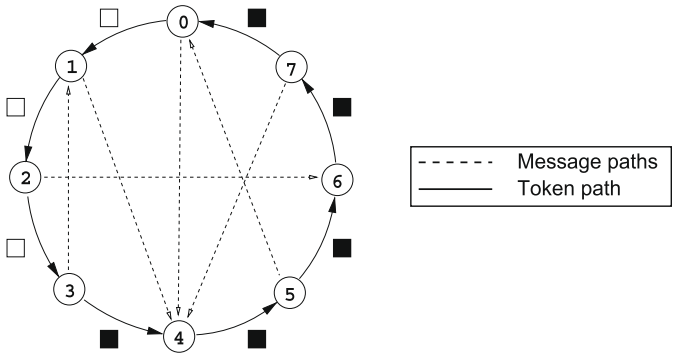


Fig. 4. Example of communication routes of Dijkstra's algorithm for a ring of eight processors. The token is sequentially passed from processor 0–7 and returns to 0. During circulation of the token, processors communicate messages related to the problem being solved.

(i) Processor P_0 always passes a white token to processor P_1 after being terminated.
(ii) When processor P_i receives the token it forwards it to P_{i+1} only after becoming inactive.
(iii) If processor P_i has sent an activation message to processor P_j, where $j < i$, then it passes a black token to P_{i+1}. If this is not the case, the token is sent to P_{i+1} with its color unchanged. Also, P_i updates the value of *Nmsg* by adding to it the difference of the number of messages it has sent minus the number of messages it has received.
(iv) If processor P_0 receives a black token or a token with $Nmsg \neq 0$ then it resends a white token to P_1 with *Nmsg* equal to the difference in number of sent/received messages of P_0 and the probing cycle is repeated. Otherwise, if the received token is white, with $Nmsg = 0$, then the algorithm terminates.

As is seen from step (iv) the status of the system is evaluated by processor P_0. A received black token indicates that several processors may still be active, while a token with $Nmsg \neq 0$ means that pending messages are *en route*. On the contrary, a received white token with $Nmsg = 0$ means that global termination has been reached. In this case P_0 scatters a message to all other processors that the algorithm has finished successfully.

4.3. Flagging of edges in parallel

In the sequel, the coupling of Dijkstra's algorithm with the adaptation algorithm presented in Section 2.4 is discussed. The desired goal is to have all local meshes properly flagged for refinement so that hanging nodes will not appear neither in their interior nor on the interpartition boundaries after refinement. A flowchart of the combined algorithm is presented in Fig. 5. Within the present context, a processor is in active state when it is applying the edge flagging rules for its elements, that is,

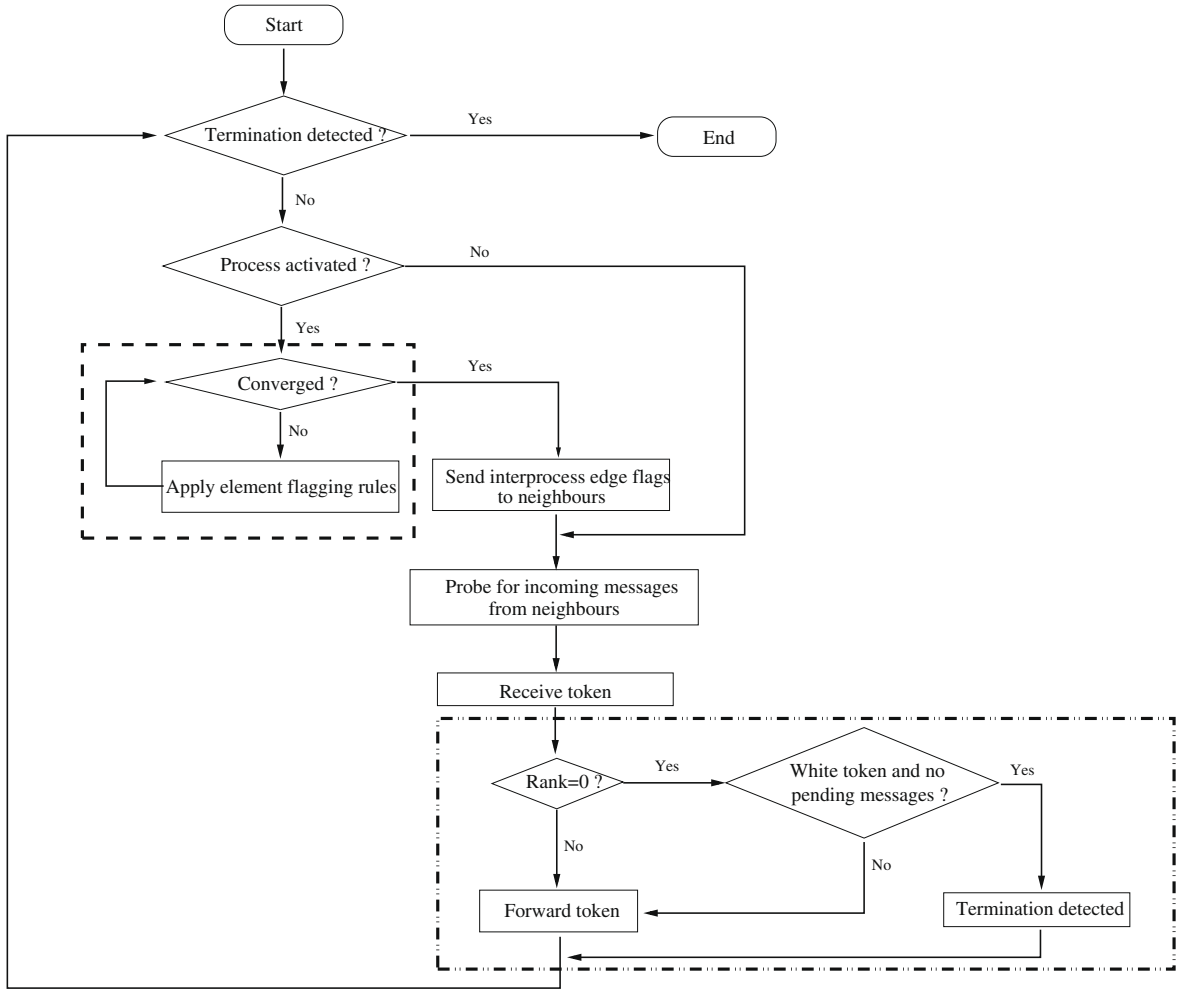


Fig. 5. Main stages of the parallel adaptation algorithm. The algorithm is composed of the serial algorithm augmented with Dijkstra's algorithm for distributed termination detection. The *while* loop for flagging of element edges (– – –) and the token communication and global termination detection (–···–) are depicted.

when executing the depicted *while* loop. For this part of the algorithm no communication with neighbours is involved. After exiting the loop, a consistently flagged local mesh has been achieved. Then, the processor is communicating the interprocessor edges' refinement marks to its face neighbours and waits for incoming messages from them (if any). The communicated messages contain triples of the form (gid_0, gid_1, f) where gid_0 and gid_1 are the global *ids* of an edge's incident vertices and f its refinement mark. In case that an otherwise unflagged interpartition edge becomes flagged by an incoming message, the processor is reactivated. As is seen, after the exchange of interpartition edges' marks, a processor receives the token from its predecessor and forwards it with the appropriate color to its successor. When global termination is detected by processor 0 all processors can safely proceed with refinement of their partitions. It is emphasized that parallel flagging of mesh edges is an *asynchronous* computation paradigm. Indeed, during its execution, processors may be in totally different states; either idle or flagging their elements' edges, or exchanging their interboundary edge' flagging marks. A *synchronous* approach (like in [27,41], for instance), where all processors complete flagging their edges and exchange edge refinement marks and then having a *master* processor to check conformity on interpartition boundaries could have been employed. The procedure is then repeated until convergence to a global mesh with no hanging nodes. Although easier to implement, this method is expected to scale worse, compared to the strategy in the present work, since it delays processors that have converged, from starting a new edge flagging iteration, until all processors have converged.

5. Dynamic load balancing

After adaptation, the mesh is appropriately redistributed among processors, before the new simulation cycle, so that performance of the numerical solution algorithm is not affected by load imbalance. The present load balancing strategy requires that (i) all sibling elements lie on the same processor as discussed before and (ii) only the original coarse mesh with proper weights is balanced. Due to this approach, coarsening and application of the isotropic refinement rule are facilitated since parent elements can be readily reinstated in the local data structures. Additionally, a significant reduction in CPU time for mesh repartitioning is observed, since the balancer [19] acts on the initial mesh. Furthermore, two complicated issues that are essential for any adaptive parallel algorithm [32] are addressed; Updating the local data structures during the phases of data migration and data reception, and determining the new communication pattern.

5.1. Exchange of data and updates of the local data structures

The load balancer identifies the elements that need to be transferred to other processors. Then, the nodes and edges that may be deleted from the local data structures are determined and boundary faces of elements tagged for migration are removed. In the sequel, processors communicate messages that contain information about the migrated elements. These messages contain data in the following order: nodes (global *ids* and coordinates), elements (*cell to node* connectivities) and boundary faces (*boundary face to node* connectivities).

After reception of a message, the global *ids* of nodes are extracted. A received node may already be present in the local data structure, if it lies on the interpartition boundary. To avoid creation of multiple copies of nodes, a temporary hash table T that initially contains the addresses of interpartition boundary nodes is defined. This temporary data structure is also useful in determining the node pointers for elements, boundary faces and edges. Utilizing table T provides a simpler and faster approach compared to [50] where balanced binary trees are employed for each type of migrated entity.

When an incoming node is extracted a search for it is performed in T. If the node does not exist in T, then a copy of it is created in the nodes' linked list and the node is hashed (stored) in T. An example of this technique is illustrated in Fig. 6. On the top of the figure, all depicted elements of processor 0 are to be migrated to processor 1. In (a), processor 1 hashes (stores) interpartition nodes A,B and C in hash table T. In (b), the received message of incoming nodes is depicted. As was seen before, the message contains only global *ids* of nodes. In step (c) all incoming nodes are hashed (searched for) in T. Nodes A,B and C are not constructed again but instances of nodes D,E and F are created, placed in the nodes' data structure and simultaneously stored in hash table T.

The second step of the data reception procedure involves the extraction of mesh elements. An incoming element is composed of the set of global *ids* of its nodes. A search operation in the hash table T yields the node pointers for mesh elements. Moreover, copies of the new elements are created and placed in the linked list of mesh elements and the edges of new elements are hashed (stored) in the edges' hash table. The incoming boundary faces are constructed in an analogous manner. Finally, edges and nodes that were flagged for deletion are removed from the local data structures.

6. Interpartition boundary updates and related topological issues

After load balancing and data migration, the new interpartition faces must be determined. Furthermore, due to topology changes after load balancing, the adjacency of processors must be re-evaluated. These two major issues are addressed in this section. In order to resolve them, a special algorithm that (i) avoids expensive searches to identify the new interpartition faces and (ii) determines the face neighbours of each processor, is applied. The interpartition boundary identification is hence composed of two phases for which efficient use of table lookup is employed. The proposed method is faster than

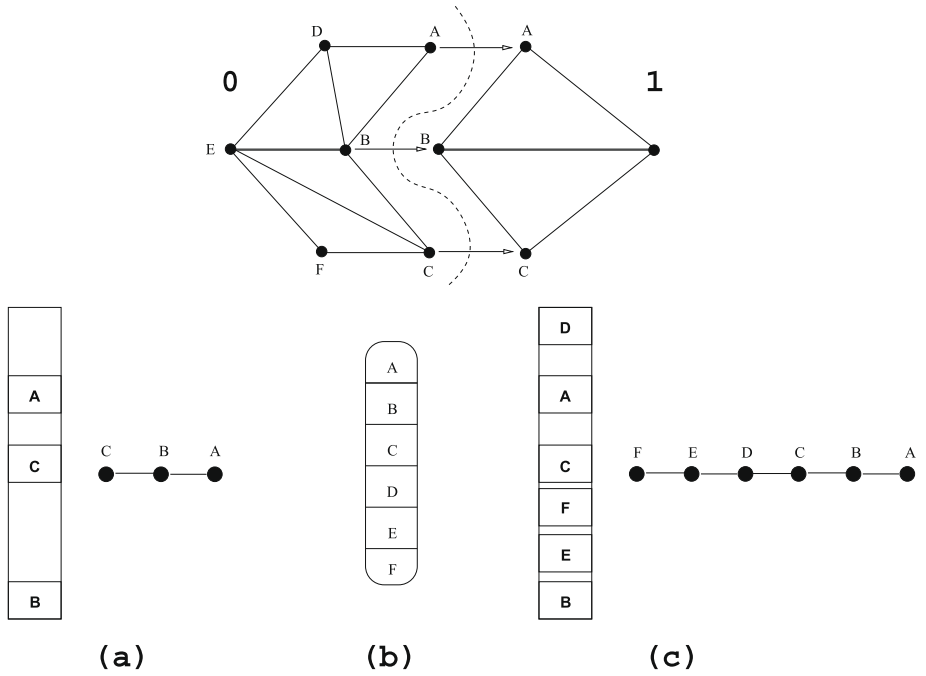


Fig. 6. Update of the local nodes data structure after data reception. Processor 1 extracts nodes from the incoming message. For nodes A, B, C it does not create any new copies, while it creates copies of nodes D, E, F in its data structure and stores them in the nodes' hash table.

the approach presented in [51] where an octree of nodes is utilized to determine interpartition boundaries and the new communication pattern.

6.1. Phase 1: determination of interpartition faces

During the first phase, processors communicate with each one of their neighbours to determine the new interpartition boundary that results from load balancing. Based only on the elements that are migrated to other processors and those received, three kinds of interpartition faces need to be identified; (i) old ones that must be deleted, (ii) old ones that also belong to the new interpartition boundary and (iii) new ones.

In the following, the algorithm for phase 1 is described, based on Figs. 7–9. An example of two processors 0 and 1 initially sharing the interpartition boundary AB (Fig. 7) is considered. Adaptation has resulted in load imbalance and elements of the shaded regions ACD and EFB are to be migrated from processor 1 to 0 and from processor 0 to 1, respectively (Fig. 8). The new interpartition boundary after load balancing is FE $\cup$ EC $\cup$ CD as shown in Fig. 9. For the moment, processor 0 is considered. The hash table of its boundary and interpartition boundary faces is denoted by T_i. The set of elements contained within region EFB and the set of the rest of elements are denoted by L_1 and L_2, respectively. Furthermore, an auxiliary hash table of faces T_a is defined.

At first, faces of the elements of L_1 are hashed (stored) in T_a and searched for in T_i. If a face is present in T_i, then it belongs to the old interpartition boundary or it is a boundary face of an element that leaves processor 0 and hence it is flagged for deletion. In the present example, processor 0 identifies faces of EB in this way. Next,the faces of the elements of L_2 are hashed (searched for) in T_a. If such a face does exist in T_a, then it belongs to the new interpartition boundary. With this approach, faces of EF are identified. In the sequel, processor 0 receives the elements of region ACD from processor 1. All faces of incoming elements are hashed (stored) in T_a. If a face is hashed twice, then it is an interior face of region ACD, otherwise, it lies on AC $\cup$ CD and is marked appropriately in T_a. Additionally, faces of received elements are hashed (searched for) in T_i. If any such face is present in T_i, then it lies on the old interpartition boundary portion AC and hence it is flagged for deletion in T_i. Moreover, it is unmarked in T_a. At the end of this procedure, AC has been eliminated and the marked faces of T_a constitute the new interpartition boundary portion CD. Using the same algorithm, processor 1 derives exactly the same set of common interpartition faces.

6.2. Phase 2: determination of face neighbour processors (new communication pattern)

On the second phase, an *all to all* communication is required to determine for each processor its new face neighbours. A processor receives the whole set of interpartition faces of every other processor and compares it with its own set. For

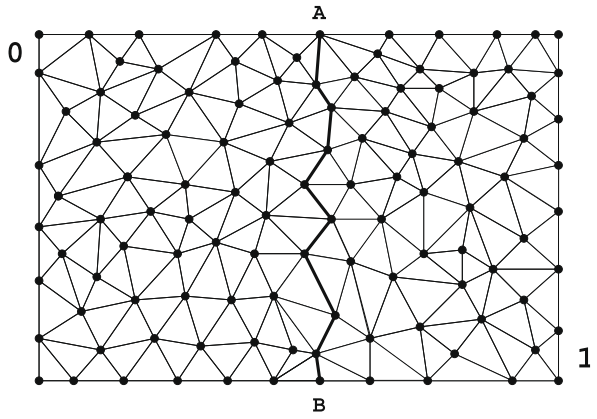


Fig. 7. Initial partition of processors 0 and 1. The interpartition boundary is AB.

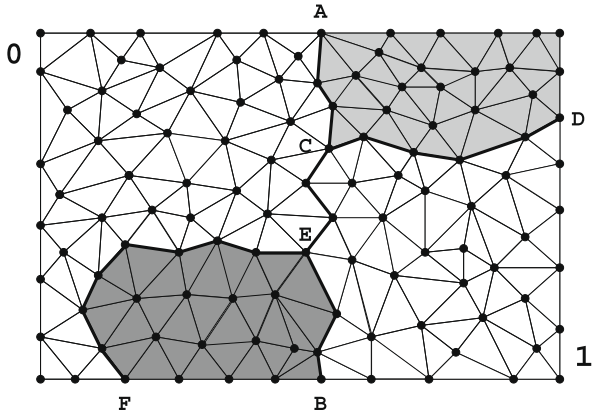


Fig. 8. Mesh elements that are flagged for migration. Elements of region ACD of processor 1 are to be transferred to processor 0. Similarly, elements of processor 0 within region EFB are flagged for migration to processor 1.

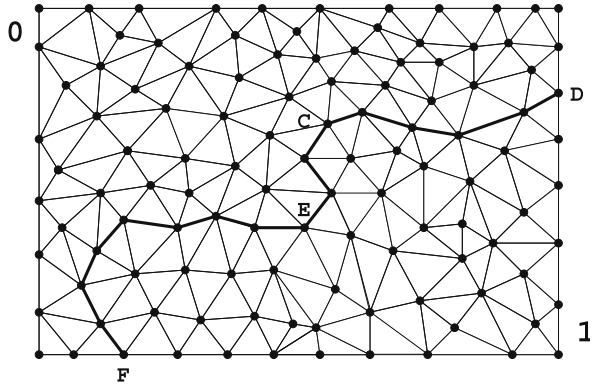


Fig. 9. New partition, after load balancing. The new interpartition boundary is the union of boundaries FE, EC and CD.

instance, suppose processor k receives the interpartition boundary S of processor l. The faces of S are hashed (searched for) in the table T_i of processor k. If any faces of S are present in T_i, then k and l are face neighbours and the face adjacency of k is adjusted.

This second phase is required to resolve the issues presented in Figs. 10 and 11. A typical situation of two processors becoming face neighbours after load balancing, although initially they are not, is presented in Fig. 10. A more difficult case is depicted in Fig. 11. As is seen, interior faces of a processor may lie on the interpartition boundary of two other processors

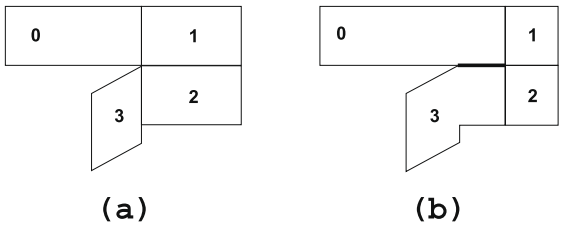


Fig. 10. In this example, processors 0 and 3 are not initially face neighbours (a). Then, processor 1 migrates elements to 0 and processor 2 transfers elements to processor 3. After load balancing (b), processors 0 and 3 share an interpartition boundary.

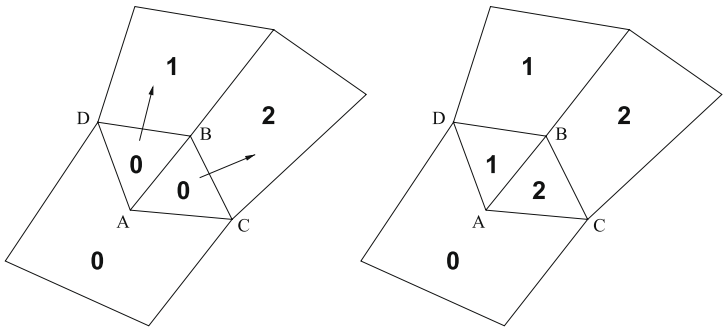


Fig. 11. In this example, elements ABD and ABC originally reside in processor 0. Then, element ABD is migrated to processor 1 and element ABC to processor 2. As a result, face AB (initially an interior face of processor 0) becomes an interpartition face of processors 1 and 2.

after load balancing. The new adjacency of processors cannot be determined during element migration by mutual communication of face neighbours only. Hence the need for this *all to all* exchange step for the resolution of these pathological cases.

7. Applications

In this section, the developed parallel adaptation scheme is applied to several representative cases. Both stationary and transient adaptation examples are considered to test the refinement and coarsening capabilities of the parallel adapter.

7.1. Global refinement of a cylinder hybrid mesh

Two levels of global refinement of a cylinder hybrid mesh are considered. The initial coarse mesh is composed of a total of 456,001 elements and contains only tetrahedra and prisms. Prisms cover the boundary layer region around the cylinder and tetrahedra fill the rest of the computational domain. The mesh is dense in the rear of the cylinder to capture the wake of a flow field. In Fig. 12 the two resulting adapted meshes after load balancing are depicted, for a run with 8 processors. As is

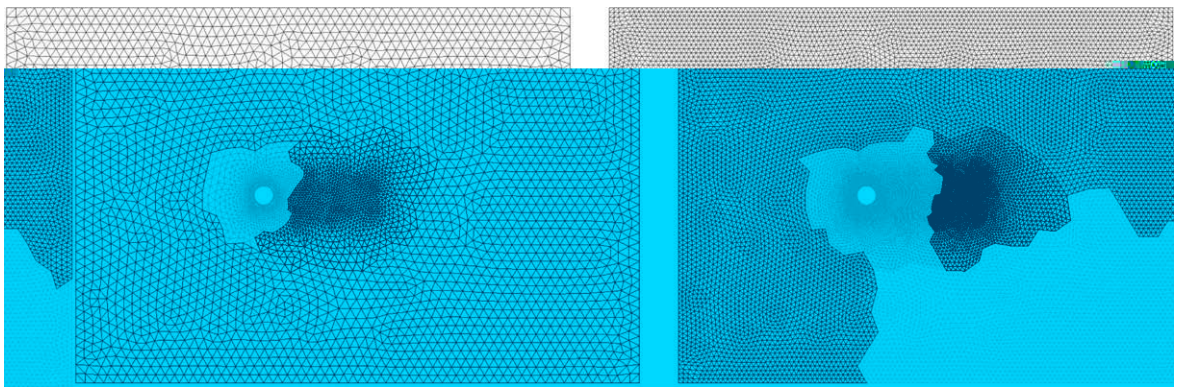


Fig. 12. Refined mesh after the first (left) and second (right) adaptation. The symmetry boundary of the computational domain and the corresponding partitions are depicted. The once refined mesh consists of 2,968,472 elements and the twice refined one of 21,029,632 elements.

seen, on the second level, the processor that occupies most of the wake region migrates elements to the processor that encompasses the cylinder. This is due to the high density of tetrahedra within the wake region and the fact that isotropic refinement of a tetrahedron produces more elements than the directional refinement of a prism. In Fig. 13 the refinement times for the two adaptations for runs up to 20 processors are presented.

7.2. Refinement and coarsening for a moving shock

The next test cases involve shock movement in a channel. At first, a channel composed of 428,868 tetrahedra is considered. At each adaptation step n, the shock occupies the region $30 + 1.75(n-1) \leqslant x \leqslant 30 + 1.75(n-1) + 2$ and this region is refined. As it can be seen, the refinement regions between successive adaptation steps overlap. The region behind the shock, $x \leqslant 30 + 1.75(n-1)$, is coarsened. A total of 10 adaptation steps are employed in a run with 8 processors. In Fig. 14 the inter-partition boundary of a partition that is adjacent to the shock region is shown. It is observed that the portion of the inter-partition boundary that touches the shock is composed of faces at levels 0, 1 and 2.

Similarly, the movement of a shock in a channel using a mesh that contains all four types of elements is simulated. The mesh is composed of 255,536 cells. Prisms and hexahedra cover the bottom and the top of the channel, respectively. The rest of the domain is tessellated with tetrahedra and pyramids connect the hexahedral/prismatic part of the mesh with the tetrahedral one. At each adaptation step n, the shock occupies the region $-250 + 20(n-1) \leqslant z \leqslant -250 + 20(n-1) + 30$ and this region is refined. The refinement regions between successive adaptation steps partially overlap and the overlapping

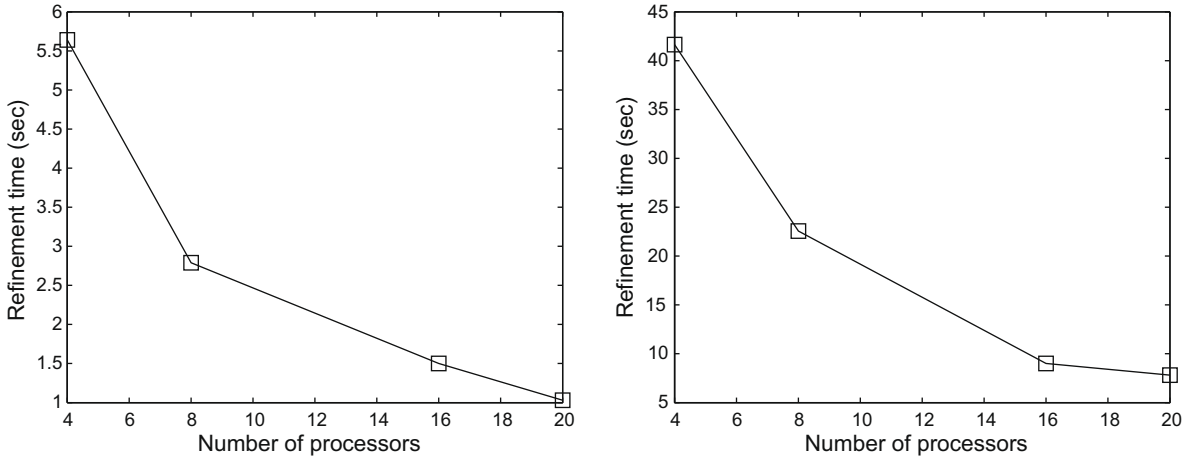


Fig. 13. Refinement time versus number of processors for the first (left) and second (right) adaptation.

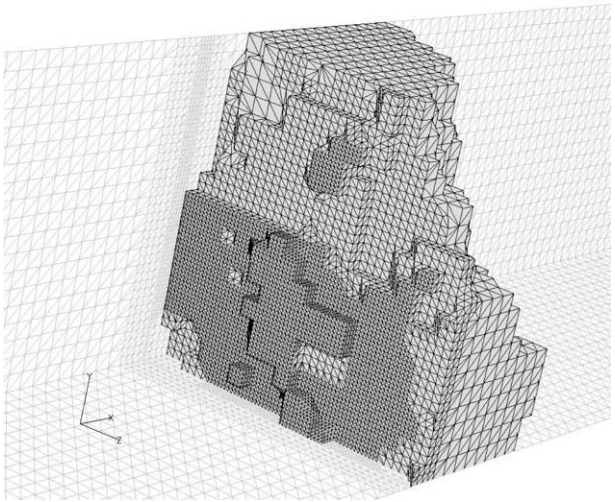


Fig. 14. Geometry of a partition that touches the moving shock.

region has length 5. The region behind the shock, $z \leqslant -250 + 20(n-1)$, is coarsened. For this run 8 processors were also used. In Fig. 15 the shock movement for 10 adaptation steps is delineated along with the corresponding partitions.

7.3. Parallel adaptation for flow around a NACA 0012 wing

The final test case is associated with subsonic turbulent flow around a wall mounted NACA 0012 wing at an angle of attack of 12°. The Reynolds number is $Re = 2 \times 10^6$ and the Mach number is $Ma = 0.22$. The original mesh contains 2,718,046 elements of all four kinds. In particular, prisms and hexahedra cover the wing and tetrahedra the rest of the domain, while pyramids form a buffer zone between the structured and unstructured portions of the mesh. The wing geometry is presented in Fig. 16 along with the flow streamlines at the free end of the wing where a vortex is created.

In the following, the parallel feature detection technique of Section 4.1 is employed. At first, one level of refinement using undivided differences of flow quantities with an *alpha* parameter $\alpha = 0.5$ is considered. The adapted mesh consists of 9,837,816 elements. In Fig. 17 the top of the wing is presented for a run with 8 processors. In column (a) it is observed that refinement is concentrated at the front of the wing to capture the boundary layer. In column (b) the original and adjusted signatures of partitions on the top of the wing are shown. As is evident, after load balancing the interpartition boundaries are shifted towards the front of the wing where more elements are refined. It should be noted that, as expected, the total numbers of elements after refinement are the same regardless of the number of processors being used.

In Figs. 18 and 19 the execution times, against number of processors, for flagging of edges using Dijkstra's algorithm and mesh refinement, respectively, are presented. The speed-up curve for refinement is shown in Fig. 20 and is fairly good since

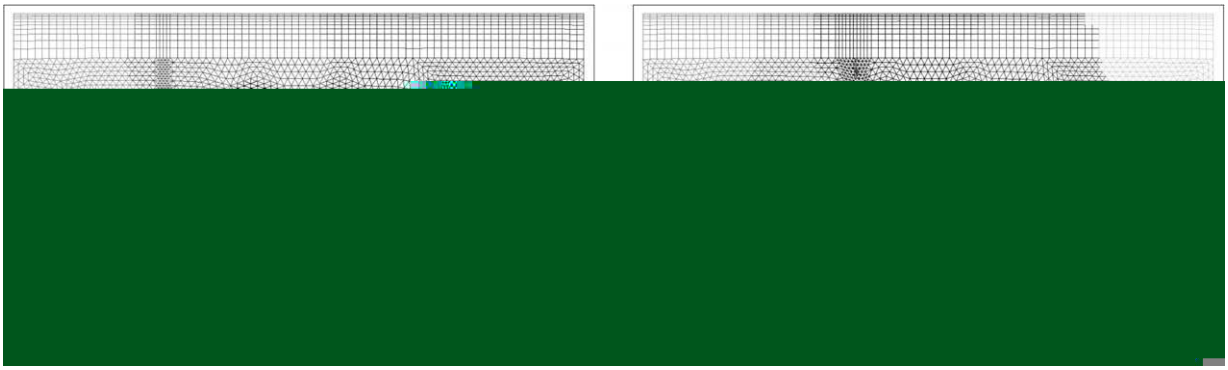


Fig. 15. Motion of a shock wave in a channel meshed with all four types of elements. The projection of the shock wave on the lateral boundary of the channel is depicted. Adaptation steps 1, 4 (left column) and 7, 10 (right column) are shown.

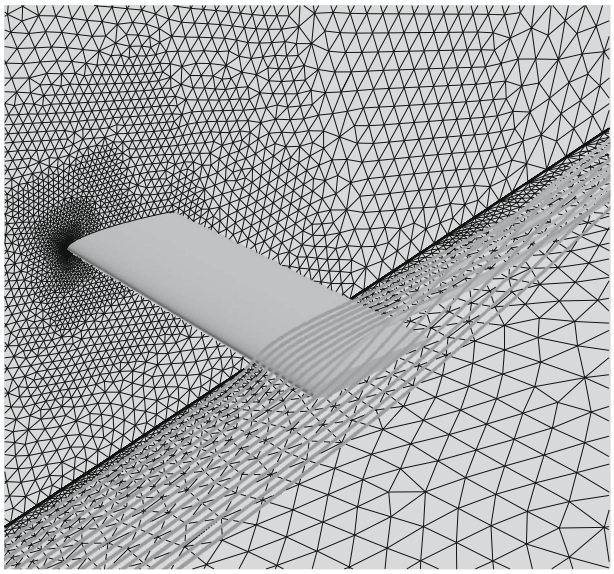


Fig. 16. NACA 0012 wing geometry and flow streamlines close to the wing tip.

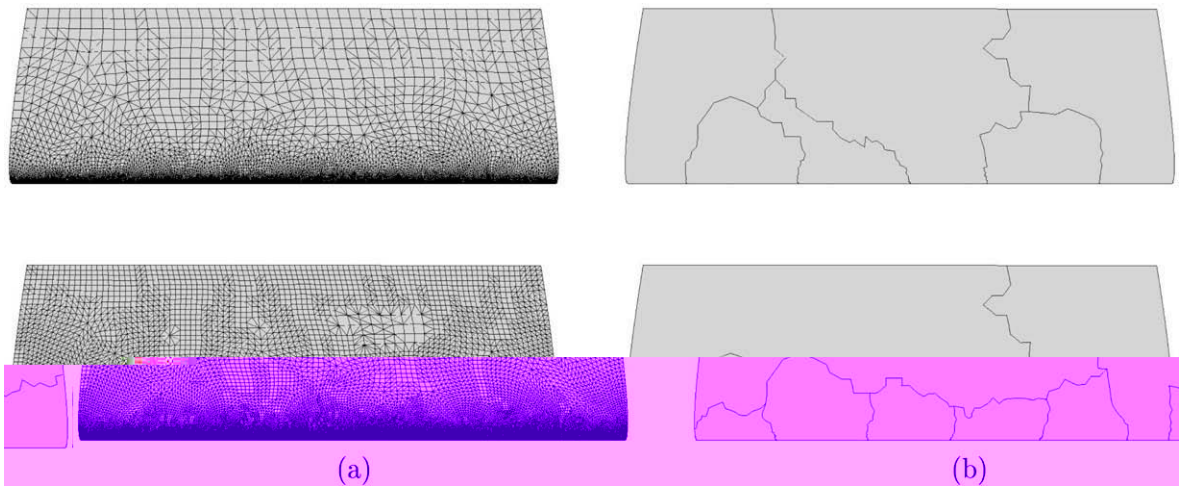


Fig. 17. Top view of the NACA 0012 wing: (a) the surface mesh before and after adaptation and (b) the corresponding partitions.

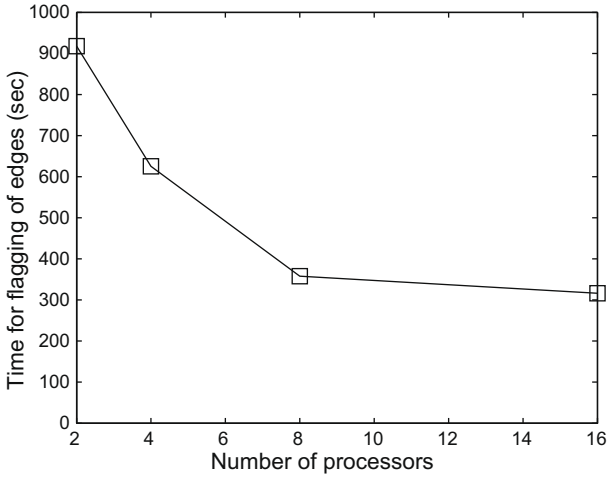


Fig. 18. Total time required for flagging of edges using Dijkstra's algorithm against number of processors. Case of subsonic flow around the NACA 0012 wing with $\alpha = 0.5$.

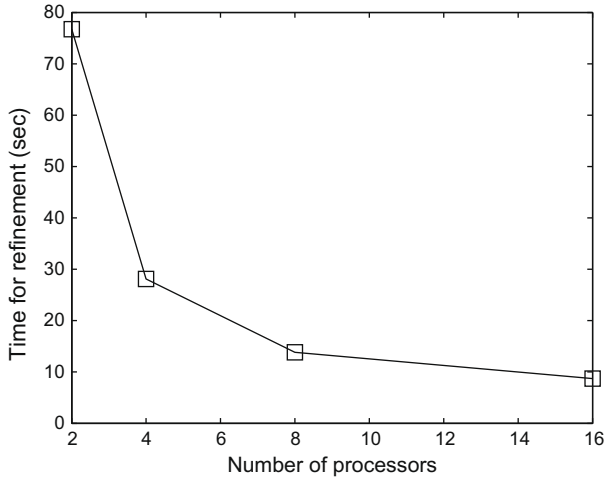


Fig. 19. Time required for mesh refinement. Case of subsonic flow around the NACA 0012 wing with $\alpha = 0.5$.

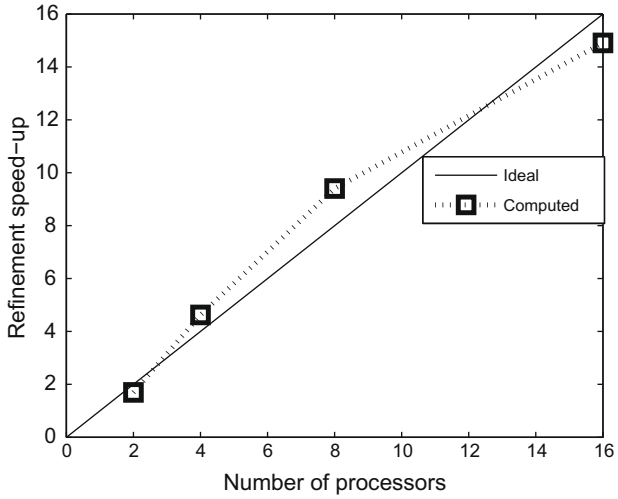


Fig. 20. Parallel speed-up for mesh refinement against number of processors. Case of subsonic flow around the NACA 0012 wing with $\alpha = 0.5$.

the partitioned mesh is balanced and no communication among processors is required during element division. In fact, super-linear speed-up is observed for 4 and 8 processors and is attributed to *Cache* memory effects. A slight drop in speed-up performance is observed for 16 processors and is due to the unbalanced distribution of edges flagged for refinement. Here, speed-up is defined as the fraction of serial execution time over parallel execution time. In Fig. 21 the speed-up for the combined edge flagging and refinement is presented. In this case, the speed-up is not close to the ideal. The reasoning for this phenomenon is that although the mesh is balanced, edges marked for refinement are not equidistributed among processors. As a result, several processors remain almost idle during the edge flagging procedure and hence the speed-up is low. Furthermore, and most importantly, the low speed-up is ascribed to the non-deterministic nature of the serial adaptation algorithm. For instance, suppose that 4 processors are employed and that the serial algorithm requires 5 iterations to conformally flag the mesh edges. If the maximum number of iterations required by one of the processors is 10, then the speed-up is 2, instead of the ideal value of 4. Additionally, a processor may get activated more than once during the execution of Dijkstra's algorithm, hence the total number of iterations to achieve a hanging node free mesh is typically much higher than that of the serial case. It would be perhaps more fair to examine speed-up using the execution time required for the 2-processor case as a reference time [21,51] (i.e. $S = \frac{T_2}{T_p}$, where T_2, T_p are the timings required for 2 and p processors, respectively), however it was desired to demonstrate parallel performance compared to the serial case. In Fig. 22 the execution times for feature detection and determination of neighbour processors are presented. It is observed that the time for feature detection is almost constant with increasing number of processors. This is expected since most of the

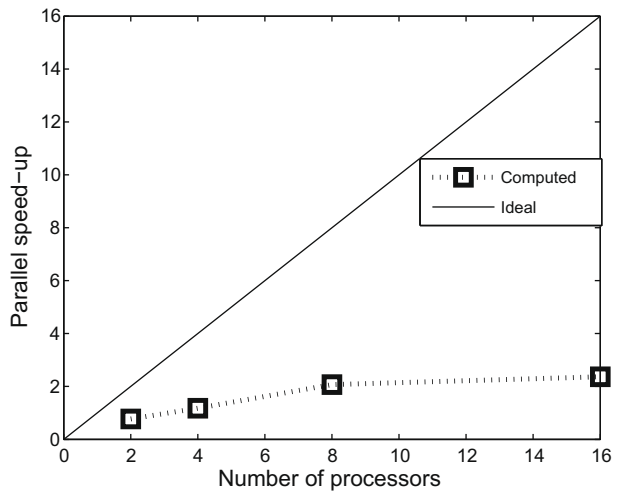


Fig. 21. Parallel speed-up for flagging of edges and mesh refinement against number of processors. Case of subsonic flow around the NACA 0012 wing with $\alpha = 0.5$.

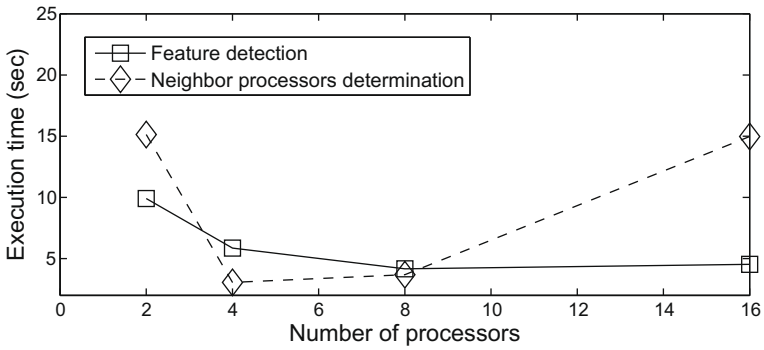


Fig. 22. Execution times for feature detection and determination of face neighbours after load balancing for the NACA 0012 wing refinement case.

computation in this case is executed by the master processor. Also, a small increase in execution time is observed for the determination of processor adjacencies with 16 processors. This phenomenon is due to the communication overhead resulting from the exchange of interpartition boundaries among processors, since an *all to all* communication step is used. Communication based on an optimal task scheduling should correct this scaling. However, it should be emphasized that the CPU time required for both operations is a minimal fraction of the total adaptation time.

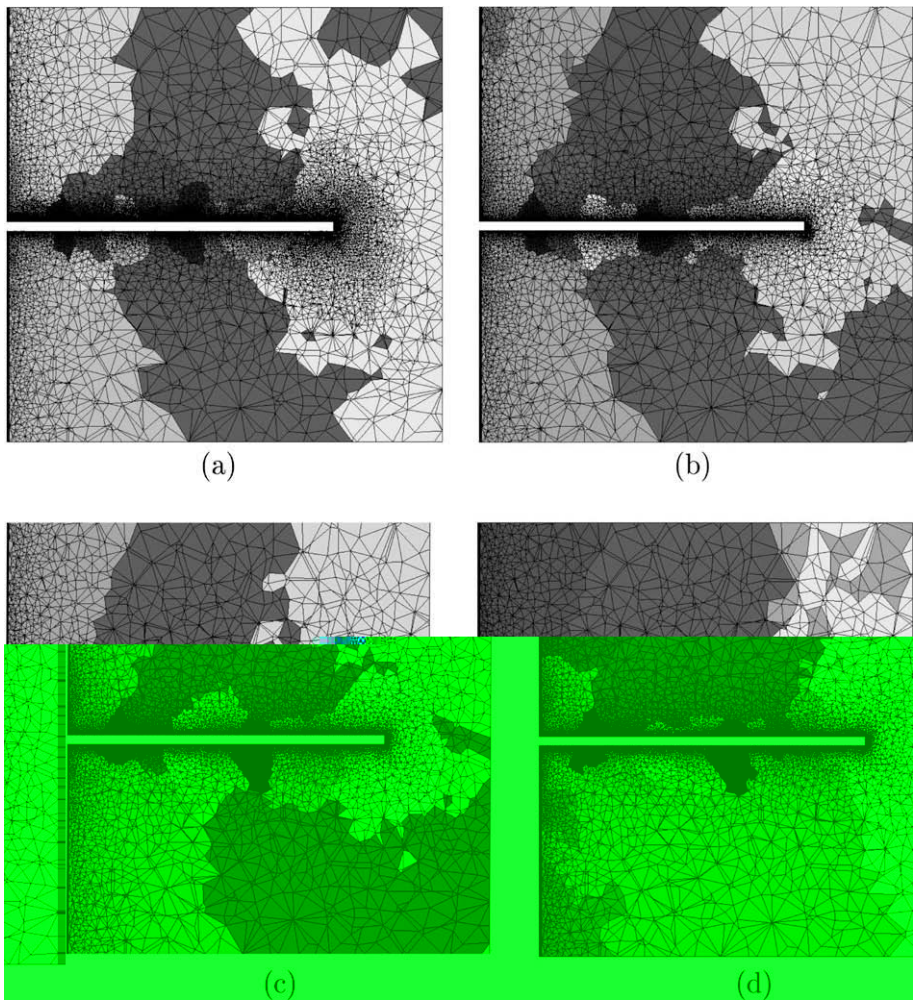


Fig. 23. Case of repeated coarsening of the NACA 0012 wing mesh with increasing values of the adaptation threshold coefficient. The adapted mesh after one level of refinement with $\alpha = 0.5$ (a), and the coarsened meshes with $\alpha = 0.7, 0.9, 1.1$. (b)–(d) are shown. The partitions on a plane cut along the wing are also depicted.

Finally, a coarsening test case for the NACA 0012 wing is considered. Having refined the mesh once, the value of the adaptation threshold coefficient α is progressively increased by a constant 0.2 and all edges with divided differences below $\mu + \alpha\sigma$ are flagged for coarsening, while edges above this threshold value are kept intact. As expected, after a large number of coarsening operations the original mesh will be obtained. Three coarsening steps for this case, with 16 processors, are considered. The total number of elements after the first refinement is 9,837,816, as in the previous case. The numbers of elements at each coarsening step are 3,640,571, 2,909,466 and 2,835,841 for $\alpha = 0.7, 0.9$ and 1.1, respectively. As is evident, the total number of elements decreases with increasing values of α. A field-cut at $x = 0.07$ of the meshes resulting first from refinement and then from successive coarsening is presented in Fig. 23. The various partitions are also delineated.

8. Summary

A parallel adaptive refinement/coarsening scheme for general three-dimensional hybrid meshes has been presented. It is based on the serial adaptation scheme of [43,46], which was parallelized on a distributed memory architecture using MPI. An attractive feature of the present work is the employment of a simple, general parallel data structure that can handle all four kinds of elements; hexahedra, prisms, pyramids and tetrahedra. In particular, an inexpensive weighted graph, associated with the coarse mesh, was used for partitioning and dynamic load balancing via the *Zoltan* library. Balancing applied to the initial coarse mesh offered speed of execution, simplicity of implementation and, in many cases, simpler and smoother interpartition boundaries. Communication for parallel exchange of edge refinement flags was based on interboundary faces and a *global to local node id* conversion hash table.

Dijkstra's distributed termination detection algorithm has been employed to implement the parallel version of the iterative edge flagging procedure. Dijkstra's algorithm respects the *asynchronous* nature of parallel edge flagging and results in a faster adaptation algorithm, compared to other studies that enforce a *synchronous* approach. Several new algorithms have also been introduced: (i) Parallel feature detection, that was based on an inexpensive use of a *master* processor, (ii) exchange of portions of local meshes and modifications of the associated data structures during data migration, (iii) determination of the processors' interpartition boundary and (iv) identification of the new communication pattern after load balancing.

Test cases considered included global refinement of a hybrid mesh for a cylinder geometry, emulation of a moving shock wave in channels meshed with tetrahedral and hybrid grids to demonstrate the coarsening capability of the present parallel adaptation algorithm and subsonic flow around a NACA 0012 wing. For the latter case, the communication pattern determination is not scalable, however its scaling can certainly be improved by using a proper task scheduling technique. Parallel feature detection is a fast process, albeit exhibiting a constant speed-up. Refinement speed-up was excellent, since no communication was involved, and in some cases super-linear. Total speed-up was particularly low due to the non-deterministic nature of the edge flagging algorithm and non-equidistribution of flagged edges.

To the best of our knowledge, this is one of the first works on parallel hierarchical refinement and coarsening of general hybrid meshes. It would be desirable to derive load balancing strategies suitable for heterogeneous architectures or for situations where the number of available processors is varying during the simulation. Future work will also include coupling of the present parallel adaptation method with various finite element and finite volume parallel CFD solvers.

References

[1] J. Waltz, Parallel adaptive refinement for unsteady flow calculations on 3D unstructured grids, International Journal for Numerical Methods in Fluids 46 (2004) 37–57.
[2] Y. Kallinderis, Hybrid grids and their applications, in: J.F. Thompson (Ed.), CRC Handbook of Grid Generation, CRC Press, Boca Raton, FL, 1999.
[3] Y. Kallinderis, A. Khawaja, H. McMorris, Hybrid prismatic/tetrahedral grid generation for viscous flows around complex geometries, Journal of the American Institute of Aeronautics and Astronautics 34 (2) (1996) 291–298.
[4] A. Basermann, J. Clinckemaillie, T. Coupez, J. Fingberg, H. Digonnet, R. Ducloux, J. Gratien, U. Hartmann, G. Lonsdale, B. Maerten, D. Roose, C. Walshaw, Dynamic load-balancing of finite element applications with the DRAMA library, Applied Mathematical Modelling 25 (2000) 83–98.
[5] M. Fiedler, Algebraic connectivity of graphs, Czechoslovak Mathematical Journal 23 (1973) 298–305.
[6] A. Pothen, H. Simon, K. Liou, Partitioning sparse matrices with eigenvectors of graphs, SIAM Journal of Matrix Analysis 11 (1990) 430–452.
[7] N. Kruyt, A conjugate gradient method for the spectral partitioning of graphs, Parallel Computing 22 (1997) 1493–1502.
[8] G. Karypis, V. Kumar, Multilevel *k*-way partitioning scheme for irregular graphs, Journal of Parallel and Distributed Computing 48 (1998) 96–129.
[9] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing 20 (1998) 359–392.
[10] B. Hendrickson, R. Leland, A Multilevel Algorithm for Partitioning Graphs, Technical Report SAND 93-1301, Sandia National Labs, 1993.
[11] A. Vidwans, Y. Kallinderis, V. Venkatakrishnan, Parallel dynamic load-balancing algorithm for three-dimensional adaptive unstructured grids, Journal of the American Institute of Aeronautics and Astronautics 32 (3) (1994) 497–505.
[12] Y. Kallinderis, Domain partitioning and load balancing for parallel computation, in: Computational Fluid Dynamics, Lecture Series 1996–2006, von Karman Institute for Fluid Dynamics, Rhode Saint Genèse, Belgium, March, 1996.
[13] T. Minyard, Y. Kallinderis, Octree partitioning of hybrid grids for parallel adaptive viscous flow simulations, International Journal for Numerical Methods in Fluids 26 (1998) 57–78.
[14] J. Flaherty, R. Loy, C. Özturan, M. Shephard, B. Szymanski, J. Teresco, L. Ziantz, Parallel structures and dynamic load balancing for adaptive finite element computation, Applied Numerical Mathematics 26 (1998) 241–263.
[15] J. Flaherty, R. Loy, M. Shephard, B. Szymanski, J. Teresco, L. Ziantz, Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws, Journal of Parallel and Distributed Computing 47 (1997) 139–152.
[16] E. Cuthill, J. McKee, Reducing the bandwidth of sparse symmetric matrices, in: Proceedings of the 24th ACM National Conference, 1969, pp. 157–172.
[17] A. Patra, J. Oden, Problem decomposition for adaptive *hp* finite element methods, Computing Systems in Engineering 6 (1995) 97–109.
[18] P. Campbell, K. Devine, J. Flaherty, L. Gervasio, J. Teresco, Dynamic Octree Load Balancing Using Space-filling Curves, Technical Report CS-03-01, Department of Computer Science, Williams College, 2003.

[19] E. Boman, K. Devine, L.A. Fisk, R. Heaphy, B. Hendrickson, V. Leung, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, Zoltan Home Page, 1999. <http://www.cs.sandia.gov/Zoltan>.
[20] B. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, The Bell System Technical Journal 29 (1970) 291–307.
[21] Y.M. Park, O.J. Kwon, A parallel unstructured dynamic mesh adaptation algorithm for 3D unsteady flows, International Journal for Numerical Methods in Fluids 48 (2005) 671–690.
[22] J. Castanos, J. Savage, Parallel refinement of unstructured meshes, in: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, MIT Boston, USA, 1999.
[23] J.G. Castanos, J.E. Savage, The dynamic adaptation of parallel mesh-based computation, in: SIAM Seventh Symposium on Parallel and Scientific Computation, 1997.
[24] T. Minyard, Y. Kallinderis, Parallel load balancing for dynamic execution environments, Computer Methods in Applied Mechanics and Engineering 189 (2000) 1295–1309.
[25] M. Shephard, J. Flaherty, C. Bottasso, H. de Cougny, C. Özturan, M. Simone, Parallel automatic adaptive analysis, Parallel Computing 23 (1997) 1327–1347.
[26] L. Oliker, R. Biswas, Plum: parallel load balancing for adaptive unstructured meshes, Journal of Parallel and Distributed Computing 52 (1998) 150–177.
[27] L. Oliker, R. Biswas, H.N. Gabow, Parallel tetrahedral mesh adaptation with dynamic load balancing, Parallel Computing 26 (2000) 1583–1608.
[28] P. Selwood, M. Berzins, P. Dew, 3D parallel mesh adaptivity: data-structures and algorithms, in: Proceedings of SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, USA, March, 1997.
[29] G. Karypis, V. Kumar, Parallel multilevel *k*-way partitioning scheme for irregular graphs, SIAM Review 41 (2) (1999) 278–300.
[30] K. Schloegel, G. Karypis, V. Kumar, A unified algorithm for load-balancing adaptive scientific simulations, Supercomputing, Dallas, Texas, 2000.
[31] Parmetis: Parallel Graph Partitioning and Fill-reducing Matrix Ordering. <http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview>.
[32] B. Hendrickson, K. Devine, Dynamic load balancing in computational mechanics, Computer Methods in Applied Mechanics and Engineering 184 (2000) 485–500.
[33] A. Vidwans, Y. Kallinderis, Unified parallel algorithm for grid adaptation on a multiple-instruction multiple-data architecture, Journal of the American Institute of Aeronautics and Astronautics 32 (9) (1994) 1800–1807.
[34] J. Remacle, J.E. Flaherty, M.S. Shephard, Parallel algorithm oriented mesh database, Engineering with Computers 18 (2002) 274–284.
[35] J. Hallberg, A. Stagg, J. Schmidt, Adaptive tetrahedral grid refinement and coarsening in message-passing environments, US Army Engineer Research and Development Center, Coastal and Hydraulics Laboratory Report. <http://chl.wes.army.mil/research/eqm/papers.htm>.
[36] H.D. Cougny, M. Shephard, Parallel refinement and coarsening of tetrahedral meshes, International Journal for Numerical Methods in Engineering 46 (1999) 1101–1125.
[37] Parmetis Manual. <http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/manual.pdf>.
[38] P. Cavallo, N. Sinha, G. Feldman, Parallel unstructured mesh adaptation for transient moving body and aeropropulsive applications, in: Proceedings of the 42nd AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, 5–8 January, 2004.
[39] P.A. Cavallo, M.J. Grismer, Further extension and validation of a parallel unstructured mesh adaptation package, in: Proceedings of the 43rd AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, 10–13 January, 2005.
[40] C. Lepage, A. St-Cyr, W.G. Habashi, Parallel unstructured mesh adaptation on distributed-memory systems, in: Proceedings of the 34th AIAA Fluid Dynamics Conference and Exhibit, Portland, Oregon, 28 June to 1 July, 2004.
[41] S. Blazy, O. Marquardt, Parallel refinement of tetrahedral meshes on distributed-memory machines, in: Proceedings of the 23rd IASTED International Multi-Conference on Parallel and Distributed Computing and Networks, Innsbruck, Austria, February 15–17, 2005.
[42] Y. Kallinderis, P. Vijayan, Adaptive refinement-coarsening scheme for three-dimensional unstructured meshes, Journal of the American Institute of Aeronautics and Astronautics 31 (1993) 1440–1446.
[43] Y. Kallinderis, C. Kavouklis, A dynamic adaptation scheme for general 3D hybrid meshes, Computer Methods in Applied Mechanics and Engineering 194 (2005) 5019–5050.
[44] E.W. Dijkstra, W. Feijen, A. van Gasteren, Derivation of a termination detection algorithm for distributed computations, Information Processing Letters 16 (1983) 217–219.
[45] E.W. Dijkstra, Shmuel Safra's Version of Termination Detection, Technical Report EWD 998, Department of Computer Science, The University of Texas at Austin, 1987.
[46] C. Kavouklis, Serial and Parallel Dynamic Adaptation of General Hybrid Meshes, Ph.D. Thesis, The University of Texas at Austin, August, 2008.
[47] Chaco: Software for Partitioning Graphs. <http://www.cs.sandia.gov/bahendr/chaco.html>.
[48] Y.G. Kallinderis, J.R. Baron, Adaptation methods for a new Navier–Stokes algorithm, Journal of the American Institute of Aeronautics and Astronautics 27 (1) (1989) 37–43.
[49] T. Baker, Mesh modification for solution adaptation and time evolving domains, in: Proceedings of the Seventh International Conference on Numerical Grid Generation in Computational Field Simulations, Whistler, British Columbia, Canada, September 25–28, 2000.
[50] C. Özturan, Distributed Environment and Load Balancing for Adaptive Unstructured Meshes, Ph.D. Thesis, Rensselaer Polytechnic Institute, August, 1995.
[51] P. Cavallo, Automated Parallel Mesh Adaptation Methods for Transient Flow Field Analyses with Fixed or Moving Boundaries, Ph.D. Thesis, Drexel University, May, 2006.