Parallel Refinement of unstructured Meshes

José G. Castaños and John E. Savage

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-99-10
July 1999

Abstract

In this paper we describe a parallel h-refinement algorithm for unstructured Finite Element meshes based on the longest-edge bisection of triangles and tetrahedrons. This algorithm is implemented in PARED, a system that supports the parallel adaptive solution of PDEs. We discuss the design of such an algorithm for distributed memory machines including the problem of propagating refinement across processor boundaries to obtain meshes that are conforming and non-degenerate. We also demonstrate that the meshes obtained by this algorithm are equivalent to the ones obtained using the serial longest-edge refinement method. We finally report on the performance of this refinement algorithm on a network of workstations.

1 Introduction

The finite-element method (FEM) is a powerful and successful technique for the numerical solution of partial differential equations. When applied to problems that exhibit highly localized or moving physical phenomena, such as occurs on the study of turbulence in fluid flows, it is desirable to compute their solutions adaptively. In such cases, adaptive computation has the potential to significantly improve the quality of the numerical simulations by focusing the available computational resources on regions of high relative error.

Unfortunately, the complexity of algorithms and software for mesh adaptation in a parallel or distributed environment is significantly greater that it is for non-adaptive computations. Because a portion of the given mesh and its corresponding equations and unknowns is assigned to each processor, the refinement (coarsening) of a mesh element might cause the refinement (coarsening) of adjacent elements some of which might be in neighboring processors. To maintain approximately the same number of elements and vertices on every processor a mesh must be dynamically repartitioned after it is refined and portions of the mesh migrated between processors to balance the work.

In this paper we discuss a method for the parallel refinement of two- and three-dimensional unstructured meshes. Our refinement method is based on Rivara's serial bisection algorithm [1, 2, 3] in which a triangle or tetrahedron is bisected by its longest edge. Alternative efforts to parallelize this algorithm for two-dimensional meshes by Jones and Plassman [4] use randomized heuristics to refine adjacent elements located in different processors.

The parallel mesh refinement algorithm discussed in this paper has been implemented as part of PARED [5, 6], a system for the parallel adaptive solution of partial differential equations that we have developed. PARED provides a variety of solvers, handles selective mesh refinement and coarsening, mesh repartitioning for load balancing, and interprocessor mesh migration. One of the most salient characteristics of adaptive code is the high sophistication of its data structures and algorithms. To cope with this complexity, new design techniques based on object-oriented technology are necessary. For this reason, all the support code in PARED is written in C++.

2 Adaptive Mesh Refinement

In the Finite Element Method (FEM) a given domain Ω is divided into a set of non-overlapping elements Ω_i such as triangles or quadrilaterals in 2D and tetrahedrons or hexahedrons in 3D. The set of elements $\Omega_i, 1 \leq i \leq n$, and its associated vertices $V_j, 1 \leq j \leq m$, form a mesh M. With the addition of boundary conditions, a set of linear equations is then constructed and solved. For the purpose of this paper, we concentrate on the refinement of *conforming unstructured* meshes composed of triangles or tetrahedrons. On unstructured meshes, a vertex can have a varying number of elements adjacent to it. Unstructured meshes are well suited to modeling domains that have

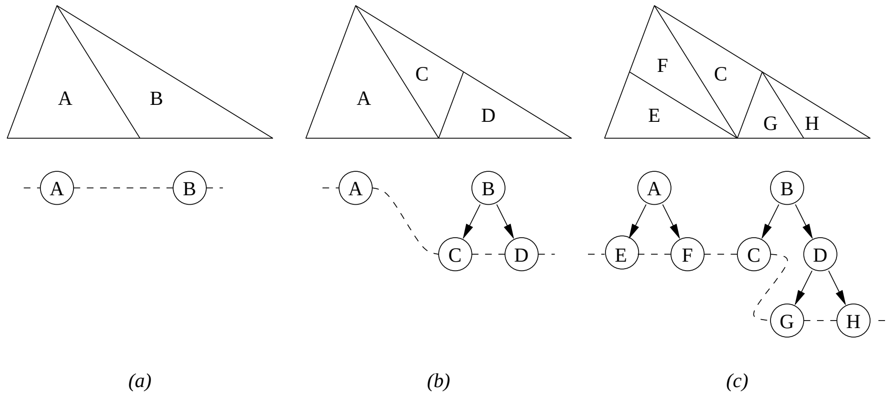


Figure 1: The refinement of the mesh in (a) using a nested refinement algorithm creates a forest of trees as shown in (b) and (c). The dotted lines identify the leaf triangles.

complex geometry. A mesh is said to be conforming if the triangles and tetrahedrons intersect only at their shared vertices, edges or faces. The FEM can also be applied to non-conforming meshes, but conformality is a property that greatly simplifies the method. It is also assumed to be a requirement in this paper.

The rate of convergence and quality of the solutions provided by the FEM depends heavily on the number, size and shape of the mesh elements. The condition number of the matrices used in the FEM and the approximation error are related to the minimum and maximum angle of all the elements in the mesh [7]. In three dimensions, the solid angle of all tetrahedrons and their ratio of the radius of the circumsphere to the inscribed sphere (which implies a bounded minimum angle) are usually used as measures of the quality of the mesh [8, 9]. For a given shape, the approximation error increases with element size (h), which is usually measured by the length of the longest edge of an element.

The goal of adaptive computation is to optimize the computational resources used in the simulation. This goal can be achieved by refining a mesh to increase its resolution on regions of high relative error in static problems or by refining and coarsening the mesh to follow physical anomalies in transient problems [10]. The adaptation of the mesh can be performed by changing the order of the polynomials used in the approximation (p-refinement), by modifying the structure of the mesh (h-refinement), or a combination of both (hp-refinement). Although it is possible to replace an old mesh with a new one with smaller elements, most h-refinement algorithms divide each element in a selected set of elements R from the current mesh into two or more nested subelements.

Starting from an initial mesh M_0 these methods construct a family of meshes $M_0, M_1, \ldots, M_L$ where every vertex in M_l is also present in M_{l+1} and every element except the ones in the initial mesh is created as the result of dividing some parent element. The numerical simulation is still based on the unrefined mesh M_L although coarser meshes are sometimes used (for example, in multigrid methods). A mesh M_l is *non-degenerate* if its interior angles are never too small or too large. An h-refinement algorithm is *stable* if, after starting from an initial mesh M_0, it never creates a degenerate mesh M_l, $0 < l \leq L$ as $L \to \infty$.

In PARED, when an element is refined, it does not get destroyed. Instead, the refined element inserts itself into a tree, where the root of each tree is an element in the initial mesh M_0 and the leaves of the trees are the unrefined elements in M_L as illustrated in Figure 1. Therefore, the refined mesh forms a forest of refinement trees. These trees are used in many of our algorithms. For example, to coarsen the mesh, we only need to replace some leaf elements in M_L by their parents. We can then repeat this procedure until we obtain the initial mesh M_0. Our repartitioning algorithm also takes advantage of these trees, and when an element is migrated to another processor all its descendants are migrated as well.

Error estimates are used to determine regions where adaptation is necessary. These estimates are obtained from previously computed solutions of the system of equations. After adaptation imbalances may result in the work assigned to processors in a parallel or distributed environment. Efficient use of resources may require that elements and vertices be reassigned to processors at runtime. Therefore, any such system for the parallel adaptive solution of PDEs must integrate subsystems for solving equations, adapting a mesh, finding a good assignment of work to processors, migrating portions of a mesh according to a new assignment, and handling interprocessor communication efficiently.

3 PARED: An Overview

PARED is a system of the kind described in the last paragraph. It provides a number of standard iterative solvers such as Conjugate Gradient and GMRES and preconditioned versions thereof. It also provides both h- and p-refinement of meshes, algorithms for adaptation, graph repartitioning using standard techniques [11] and our own *Parallel Nested Repartitioning* (PNR) [6, 12], and work migration.

PARED runs on distributed memory parallel computers such as the IBM SP-2 and networks of workstations. These machines consist of coarse-grained nodes connected through a high to moderate latency network. Each node cannot directly address a memory location in another node. In PARED nodes exchange messages using MPI (Message Passing Interface) [13, 14, 15]. Because each message has a high startup cost, efficient message passing algorithms must minimize the number of messages delivered. Thus, it is better to send a few large messages rather than many small ones. This is a very important constraint and has a significant impact on the design of message passing algorithms.

PARED can be run interactively (so that the user can visualize the changes in the mesh that results from mesh adaptation, partitioning and migration) or without direct intervention from the user. The user controls the system through a GUI in a distinguished node called the *coordinator*, P_C. This node collects information from all the other processors (such as its elements and vertices). This tool uses OpenGL [16] to permit the user to view 3D meshes from different angles. Through the *coordinator*, the user can also give instructions to all processors such as specifying when and how to adapt the mesh or which strategy to use when repartitioning the mesh.

When the FEM is implemented in parallel, a portion of the mesh is assigned to each processor. The goal is to find a partition of the mesh that minimizes interprocessor communication while assigning similar amount of work to each processor. This goal is related to the graph partitioning problem [17], an NP-complete problem for which many heuristics have been developed [18, 19]. To simplify the way matrices are computed in the FEM, it is preferred to make this partition by mesh elements rather than by mesh vertices [20] so that matrices can be assembled locally when creating a systems of equations. A partition of the mesh can be achieved by computing the dual of the mesh, that is, a graph in which there is a vertex for every mesh element and an edge between neighboring elements. This graph can then be partitioned using any of a number of graph partitioning algorithms.

In our computation, we assume that an initial coarse mesh is given and that it is loaded into the coordinator. The initial mesh can then be partitioned using one of a number of serial graph partitioning algorithms and distributed between the processors. PARED then starts the simulation.

Based on some *adaptation criterion* [21], PARED adapts the mesh using the algorithms explained

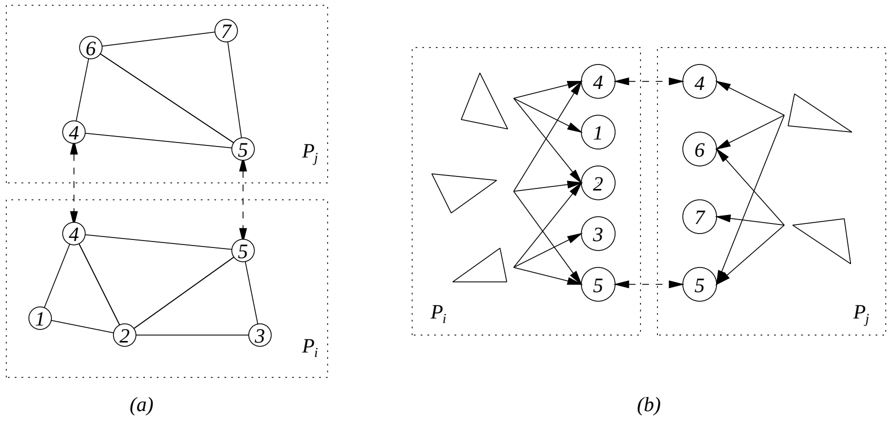


Figure 2: Mesh representation in a distributed memory machine using remote references.

in Section 5. After the adaptation phase, PARED determines if a workload imbalance exists due to increases and decreases in the number of mesh elements on individual processors. If so, it invokes a procedure to decide how to repartition mesh elements between processors; and then moves the elements and vertices. We have found that PNR gives partitions with a quality comparable to those provided by standard methods such as Recursive Spectral Bisection [18] but which handles much larger problems than can be handled by standard methods.

3.1 Object-Oriented Mesh Representations

In PARED every element of the mesh is assigned to a unique processor. Vertices are shared between two or more processors if they lie on a boundary between partitions. Each of these processors has a copy of the shared vertices and vertices refer to each other using remote references, a concept used in object-oriented programming. This is illustrated in Figure 2 on which the remote references (marked with dashed arrows) are used to maintain the consistency of multiple copies of the same vertex in different processors. Remote references are functionally similar to standard C pointers but they address objects in a different address space. When implemented in C++, a remote reference is an object that consists of a processor number and memory address.

A processor can use remote references to invoke methods on objects located in a different processor. In this case, the method invocations and arguments destined to remote processors are marshaled into messages that contain the memory addresses of the remote objects. In the destination processors these addresses are converted to pointers to objects of the corresponding type through which the methods are invoked. Because the different nodes are inherently trusted and MPI guarantees reliable communication, PARED does not incur the overhead traditionally associated with distributed object systems.

Another idea commonly found in object oriented programming and which is used in PARED is that of smart pointers. An object can be destroyed when there are no more references to it. In PARED vertices are shared between several elements and each vertex counts the number of elements referring to it. When an element is created, the reference count of its vertices is incremented. Similarly, when the element is destroyed, the reference count of its vertices is decremented. When the reference count of a vertex reaches zero, the vertex is no longer attached to any element located in the processor and it can be destroyed. If a vertex is shared, then some processor might have a

Bisect(Ω_i)
let V_p, V_q and V_r be vertices of the triangle Ω_i
let (V_p, V_q) be the longest side of Ω_i and let V_s be the midpoint of (V_p, V_q)
bisect Ω_i by the edge (V_r, V_s), generating two new triangles Ω_{i_1} and Ω_{i_2}
while V_s is a non-conforming vertex **do**
  find the non-conforming triangle Ω_j adjacent to the edge (V_p, V_q)
  Bisect(Ω_j)
end while

Figure 3: Longest edge (Rivara) bisection algorithm on triangular mesh.

remote reference to it. In that case, before a copy of a shared vertex is destroyed, it informs the copies in other processors to delete their references to itself. This procedure insures that the shared vertex can then be safely destroyed without leaving dangerous dangling pointers referring to it in other processors.

Smart pointers and remote references provide a simple replication mechanism that is tightly integrated with our mesh data structures. In adaptive computation, the structure of the mesh evolves during the computation. During the adaptation phase, elements and vertices are created and destroyed. They may also be assigned to a different processor to rebalance the work. As explained above, remote references and smart pointers greatly simplify the task of creating dynamic meshes.

4 Adaptation Using the Longest Edge Bisection Algorithm

Many h-refinement techniques [22] have been proposed to serially refine triangular and tetrahedral meshes. One widely used method is the longest-edge bisection algorithm proposed by Rivara [1, 2]. This is a recursive procedure (see Figure 3) that in two dimensions splits each triangle Ω_i from a selected set of triangles R by adding an edge between the midpoint V_s of its longest side to the opposite vertex. In the case that V_s makes a neighboring triangle, Ω_j, non-conforming, then Ω_j is refined using the same algorithm. This may cause the refinement to propagate throughout the mesh, as illustrated in Figure 4. Nevertheless, this procedure is guaranteed to terminate because the edges it bisects increase in length. Building on the work of Rosenberg and Stenger [23] on bisection of triangles, Rivara [1, 2] shows that this refinement procedure provably produces two dimensional meshes in which the smallest angle of the refined mesh is no less than half of the smallest angle of the original mesh.

The longest-edge bisection algorithm can be generalized to three dimensions [3] where a tetrahedron is bisected into two tetrahedrons (Figure 5) by inserting a triangle between the midpoint of its longest edge and the two vertices not included in this edge. The refinement propagates to neighboring tetrahedra in a similar way. This procedure is also guaranteed to terminate, but unlike the two dimensional case, there is no known bound on the size of the smallest angle. Nevertheless, experiments conducted by Rivara [3] suggest that this method does not produce degenerate meshes.

In two dimensions there are several variations on the algorithm. For example a triangle can initially be bisected by the longest edge, but then its children are bisected by the non-conforming edge, even if it is that is not their longest edge [1]. In three dimensions, the bisection is always performed by the longest edge so that matching faces in neighboring tetrahedrons are always bisected by the same common edge.

Because in PARED refined elements are not destroyed in the refinement tree, the mesh can be

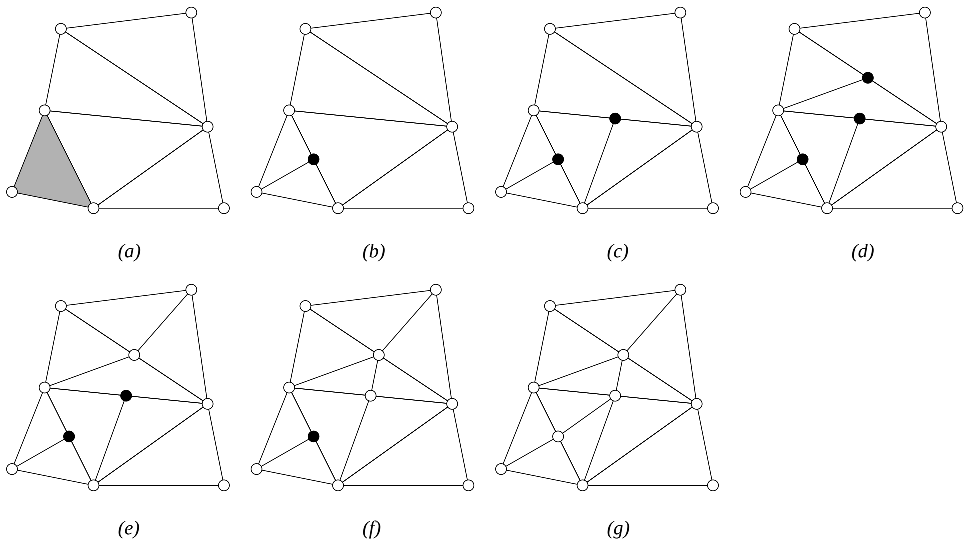


Figure 4: The refinement of the shaded triangle *(a)* propagates through the mesh creating three (black) non-conforming vertices *(b)-(f)*. Non-conforming elements adjacent to these vertices are recursively refined until all the mesh is conforming (g).

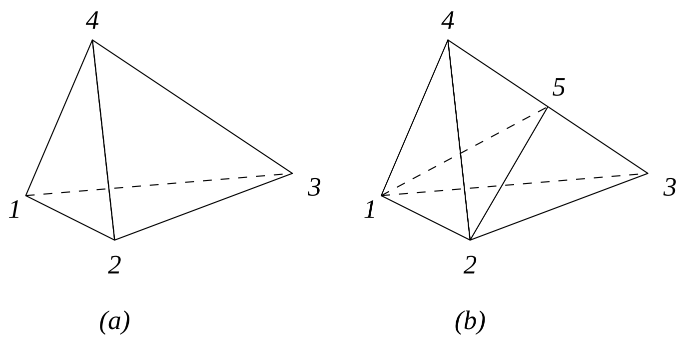


Figure 5: Refinement of a tetrahedron *(a)* by its longest edge $(3,4)$ into two tetrahedrons (b). The refinement propagates to all the tetrahedrons adjacent to the edge $(3,4)$.

coarsened by replacing all the children of an element by their parent. If a parent element Ω_i is selected for coarsening, it is important that all the elements Ω_j that are adjacent to the longest edge of Ω_i are also selected for coarsening. If neighbors are located in different processors then only a simple message exchange is necessary. This algorithm generates conforming meshes: a vertex is removed only if all the elements that contain that vertex are all coarsened. It does not propagate like the refinement algorithm and it is much simpler to implement in parallel. For this reason, in the rest of the paper we will focus on the refinement of meshes.

5 Parallel Longest-Edge Refinement

The initial coarse mesh is assumed small enough so it can be stored in one processor and refined there using the serial refinement algorithms outlined in the previous section. Nevertheless, as the number of elements and vertices increases in refined meshes, it is necessary to distribute the mesh between processors and perform its refinement in parallel.

Because the adaptation procedure is used to refine the mesh in regions of high relative error, when the mesh is distributed between processors it is likely that such a region is located in only one or few processors. The longest edge bisection algorithm and many other mesh refinement algorithms that propagate the refinement to guarantee conformality of the mesh are not local. The refinement of one particular triangle or tetrahedron Ω_i can propagate through the mesh and potentially cause changes in regions far removed from Ω_i. If neighboring elements are located in different processors, it is necessary to propagate this refinement across processor boundaries to maintain the conformality of the mesh.

In our parallel longest edge bisection algorithm each processor P_i iterates between a serial phase, in which there is no communication, and a parallel phase, in which each processor sends and receives messages from other processors. In the serial phase, processor P_i selects a set R_i of its elements for refinement and refines them using the serial longest edge bisection algorithms outlined earlier. The refinement often creates shared vertices in the boundary between adjacent processors. P_i maintains a list for each of its neighboring processors P_j of the vertices shared with that processor that were created during the serial phase. To minimize the number of messages exchanged between P_i and P_j, P_i delays the propagation of refinement to P_j until P_i has refined all the elements in R_i.

The serial phase terminates when P_i has no more elements to refine. When this happens, P_i sends one message to every adjacent processor P_j on whose shared boundary it created one or more shared vertices. This message contains all the vertices shared with P_j that were created as a result of refining the elements in R_i. After sending these messages P_i listens for messages from other processors.

A processor P_i informs an adjacent processor P_j that some of its elements need to be refined by sending a message from P_i to P_j containing the non-conforming edges and the vertices to be inserted at their midpoint. Each edge is identified by its endpoints V_p and V_q and its remote references (see Figure 6 (a)). If V_p and V_q are shared vertices, then P_i has a remote reference to copies of V_p and V_q located in processor P_j. These references are included in the message, so that P_j can identify the non-conforming edge e and insert the new vertex V_s. A similar strategy can be used when the edge is refined several times during the refinement phase, but in this case, the vertex V_s is not located at the midpoint of e.

Different processors can be in different phases during the refinement. For example, at any given time a processor can be refining some of its elements (serial phase) while neighboring processors have refined all their elements and are waiting for propagation messages (parallel phase) from adjacent processors. P_j waits until it has no elements to refine before receiving a message from P_i. For every

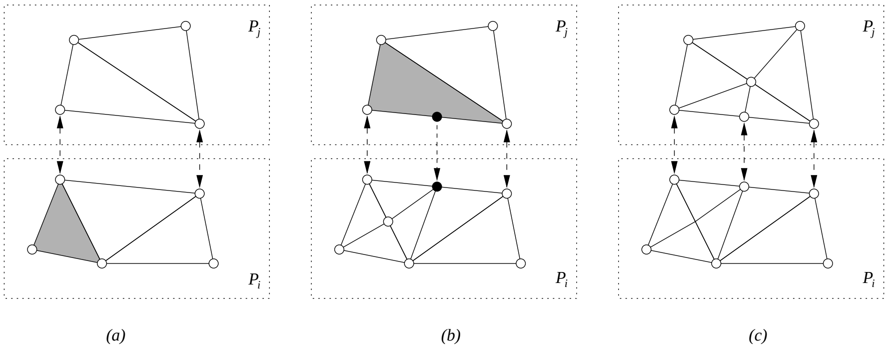


Figure 6: (a) In the parallel longest edge bisection algorithm some elements (shaded) are initially selected for refinement. (b) If the refinement creates a new (black) vertex on a processor boundary, the refinement propagates to neighbors. (c) Finally the references are updated accordingly.

nonconforming edge e included in a message to P_j, P_j creates its shared copy of the midpoint V_s (unless it already exists) and inserts the new non-conforming elements adjacent to V_s into a new set R'_j of elements to be refined. In this way the refinement propagates across the internal boundary between P_i and P_j to maintain the conformality of the global mesh. The remote references of the new vertices are updated accordingly. The copy of V_s in P_j must also have a remote reference to the copy of V_s in P_i. For this reason, when P_i propagates the refinement to P_j it also includes in the message a reference to its copies of shared vertices. These steps are illustrated in Figure 6. P_j then enters the serial phase again, where the elements in R'_j are refined.

The parallel algorithm is essentially the same for two- and three-dimensional meshes. In both cases triangles and tetrahedrons are refined by their longest edge. The only difference is that in 2D, every triangle can have only one neighboring triangle over a shared edge so it needs to propagate the refinement to only one processor. Because a tetrahedron can have more than one neighboring tetrahedron over an edge and these tetrahedra can be located in different processors, the creation of a shared vertex in a 3D mesh might occur between one processor and several other processors.

5.1 The Challenge of Refining in Parallel

The description of the parallel refinement algorithm is not complete because refinement propagation across processor boundaries can create two synchronization problems. The first problem, *adaptation collision*, occurs when two (or more) processors decide to refine adjacent elements (one in each processor) during the serial phase, creating two (or more) vertex copies over a shared edge, one in each processor. It is important that all copies refer to the same logical vertex because in a numerical simulation each vertex must include the contribution of all the elements around it.

The second problem that arises, *termination detection*, is the determination that a refinement phase is complete. The serial refinement algorithm terminates when the processor has no more elements to refine. In the parallel version, a processor might have no elements to refine but it might still be waiting for refinement propagations from adjacent processors. Therefore, termination is a global decision that cannot be determined by an individual processor and requires a collaborative effort of all the processors involved in the refinement. We now describe these two problems and our solutions.

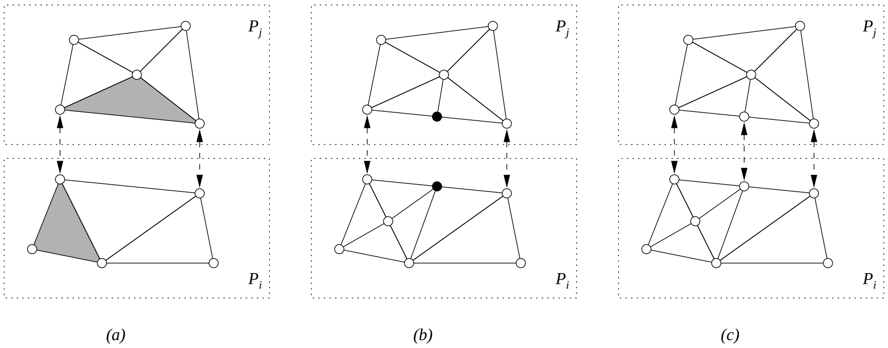


Figure 7: (a) Both processors select (shaded) mesh elements for refinement. The refinement propagates to a neighboring processor (b) resulting in more elements being refined (c).

5.1.1 Adaptation collision

During the serial phase, every processor P_i refines a selected set R_i of its elements in parallel (these elements are shaded in the Figure 7). In the event that two or more processors P_i and P_j both refine an edge e that is shared between mesh elements located in different processors, they create a vertex over the shared edge before propagating the refinement to the others. In this case it is important that these processors do not consider these vertices to be distinct.

When a message arrives at P_j with the instruction to create a vertex V_r at the midpoint of a shared edge e, P_j can detect if the edge was already refined by using e to check the elements in P_j that contain it. If any of those elements was refined over e then P_j already has a copy of the vertex V_r. In that case, P_j does not create another copy of the same vertex but instead it adds to it a reference to the remote copy in P_i. As we mentioned earlier, this reference is included in the message from P_i to P_j. Otherwise, P_j creates the vertex V_r and then refines the non-conforming elements adjacent to it after which it adds a reference to that remote copy in P_i.

5.1.2 Termination detection

The other problem mentioned earlier is the detection of the termination of refinement. Termination of refinement cannot be determined by individual processors because, although a processor P_i may have adapted all of its mesh elements in R_i, it cannot determine whether this condition holds for all other processors. For example, at any given time, no processor might have any more elements to refine. Nevertheless, the refinement cannot terminate because there might be some propagation messages in transit.

The algorithm for detecting the termination of parallel refinement is based on Dijkstra's general distributed termination algorithm [24, 25]. A global termination condition is reached when no element is selected for refinement. Hence if R is the set of all elements in the mesh currently marked for refinement, then the algorithm finishes when $R = \emptyset$.

The termination detection procedure uses message acknowledgments. For every propagation message that P_j receives, it maintains the identity of its source (P_i) and to which processors P_k it propagated refinements. Each propagation message is acknowledged. P_j acknowledges to P_i after it has refined all the non-conforming elements created by P_i's message and has also received acknowledgments from all the processors to which it propagated refinements.

A processor P_i can be in two states: an inactive state is one in which P_i has no elements to

for each processor P_i **do**
  P_C sends a *RefineMsg* to P_i specifying elements to refine and/or an adaptation criterion
end for
for each processor P_i **do**
  P_C waits for a *DoneMsg* from P_i
end for
for each processor P_i **do**
  P_C sends a *FinishMsg* to P_i to indicate the termination of the refinement
end for

Figure 8: The messaging algorithm executed by the coordinator P_C.

refine (it cannot send new propagation messages to other processors) but can receive messages. If P_i receives a propagation message from a neighboring processor, it moves from an inactive state to an active state, selects the elements for refinement as specified in the message and proceeds to refine them. Let R_i be the set of elements in P_i needing refinement. A processor P_i becomes inactive when:

- P_i has received an acknowledgment for every propagation message it has sent.
- P_i has acknowledged every propagation message it has received.
- $R_i = \emptyset$.

Using this definition, a processor P_i might have no more elements to refine ($R_i = \emptyset$) but it might still be in an active state waiting for acknowledgments from adjacent processors. When a processor becomes inactive, P_i sends an acknowledgment to the processors whose propagation message caused P_i to move from an inactive state to an active state.

We assume that the refinement is started by the coordinator processor, P_C (see Figure 8). At this stage, P_C is in the active state while all the processors are in the inactive state. P_C initiates the refinement by sending the appropriate messages to other processors. This message also specifies the adaptation criterion to use to select the elements R_i for refinement in P_i.

When a processor P_i receives a message from P_C, it changes to an active state, selects some elements for refinement either explicitly or by using the specified adaptation criterion, and then refines them using the serial bisection algorithm, keeping track of the vertices created over shared edges as described earlier. When it finishes refining its elements, P_i sends a message to each processor P_j on whose shared edges P_i created a shared vertex. P_i then listens for messages.

Only when P_i has refined all the elements specified by P_C and is not waiting for any acknowledgment message from other processors does it sends an acknowledgment to P_C. Global termination is detected when the coordinator becomes inactive. When P_C receives an acknowledgment from every processor this implies that no processor is refining an element and that no processor is waiting for an acknowledgment. Hence it is safe to terminate the refinement. P_C then broadcasts this fact to all the other processors.

5.2 The Message Model for Refinement

The parallel refinement algorithm described above uses three different types of message:

- A *RefineMsg* received by P_i indicates that some elements in P_i must be refined. At the beginning of the refinement phase, P_C sends a *RefineMsg* to all the processors. The processors

```
while true do
  P_i waits for a message msg from other processors (including P_C)
  if msg type = FinalizeMsg then
    break
  else if msg type = RefineMsg then
    refine the specified elements using a serial refinement algorithm
    if refinement does not propagate to other processors then
      send a DoneMsg to the source processor of msg
    else
      send a RefineMsg to every processor to which the refinement propagates
    end if
  else if msg type = DoneMsg then
    let msg be a response to a previous RefineMsg from P_i to P_j. Assume that the refinement
    that propagated to P_j was caused by a message received by P_i from P_k
    if P_i is waiting for no more responses to RefineMsg then
      send a DoneMsg to P_k
    end if
  end if
end while
```

Figure 9: Message model for refinement by processor P_i.

select their elements to refine according to the specified refinement criterion. This message is also used when the refinement propagates between the processors. In this case, the message indicates which edge to refine and a reference to the new vertices.

- A *DoneMsg* is used to acknowledge each *RefineMsg* received by P_i from another processor P_j (including the coordinator).
- A *FinishMsg* from P_C to each processors signals the end of the refinement phase.

The steps executed by the coordinator P_C and by the other processors P_i are different. Figure 8 shows the algorithm executed by the coordinator. P_C initially broadcasts a *RefineMsg* to all the other processors to start the refinement phase which identifies the initial elements to refine. P_C then waits until it receives a *DoneMsg* from all the other processors. At this point, P_C broadcasts a *FinishMsg*.

Every other processor P_i is in a loop waiting for messages (Figure 9). P_i does not know which type of message it is going to receive next, so it has to be able to receive any of them. This algorithm is nondeterministic because at any given time P_i can receive a message from any of its neighbors or the coordinator. In MPI, messages received from the same source are received in order, but there is no guarantee about the order of messages from different sources. Therefore in different executions the sequence in which messages are received can be different.

If P_i receives a message whose type is a *RefineMsg* from the coordinator, then P_i selects a set of its elements for refinement and refines them using the serial bisection algorithm. If the *RefineMsg* is from another processor P_j, then for each edge specified in the message, P_i creates or updates the reference to the corresponding vertex of that edge and then refines the adjacent non-conforming elements. If the refinement of the elements specified in the *RefineMsg* creates vertices over edges shared with adjacent processors, then P_i sends a corresponding *RefineMsg* to every such processor. Otherwise it sends a *DoneMsg* to the source of the *RefineMsg*.

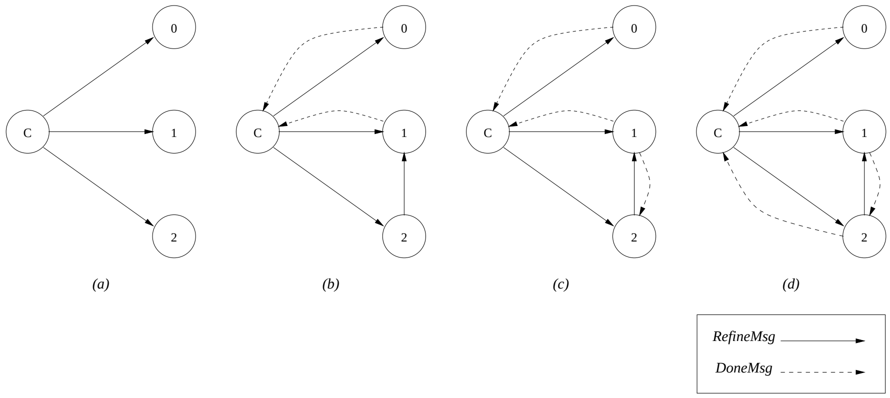


Figure 10: (a) P_C sends a *RefineMsg* to all processors. (b) Because P_0 and P_1 do not propagate the refinement to other processors they return a *DoneMsg* to P_C. P_2 propagates the refinement to P_1. (c) P_1 refines the nonconforming elements in P_1 without propagating the refinement to other processors and then sends a *DoneMsg* to P_2. (d) A *DoneMsg* is sent from P_2 to P_C.

As mentioned earlier, every *RefineMsg* must be acknowledged by a *DoneMsg* which is returned to the source of the *RefineMsg* when P_i has completed all the propagations caused by that message. This is true when

- P_i has refined all its local elements specified in the *RefineMsg* message, and
- P_i has also received a *DoneMsg* from every other processor to which each refinement has propagated.

P_i stops waiting for messages when it receives a *FinalizeMsg* from P_C. As suggested in Figure 10 (b), a processor might have returned a *DoneMsg* to the coordinator but might still have more elements to refine because it received a *RefineMsg* from an adjacent processor (P_2 in this case).

When P_i propagates refinements to P_j, P_i does not block while waiting for an acknowledgment from P_j. Instead it waits in a loop accepting messages from other processors. A deadlock is very likely to occur if P_i were to block. This scheme, propagating the refinement and listening for messages, allows us to handle cases like the one shown in Figure 11 in which the refinement initiated in P_i propagates back to the same processor. In this example, the processors denoted P_j and P_k in Figure 9 are actually the same.

6 Properties of Meshes Refined in Parallel

Our parallel refinement algorithm is guaranteed to terminate. In every serial phase the longest edge bisection algorithm is used. In this algorithm the refinement propagates towards progressively longer edges and will eventually reach the longest edge in each processor. Between processors the refinement also propagates towards longer edges. Global termination is detected by using the global termination detection procedure described in the previous section. The resulting mesh is conforming. Every time a new vertex is created over a shared edge, the refinement propagates to adjacent processors. Because every element is always bisected by its longest edge, for triangular meshes the results by Rosenberg and Stenger on the size of the minimum angle of two-dimensional meshes also hold.

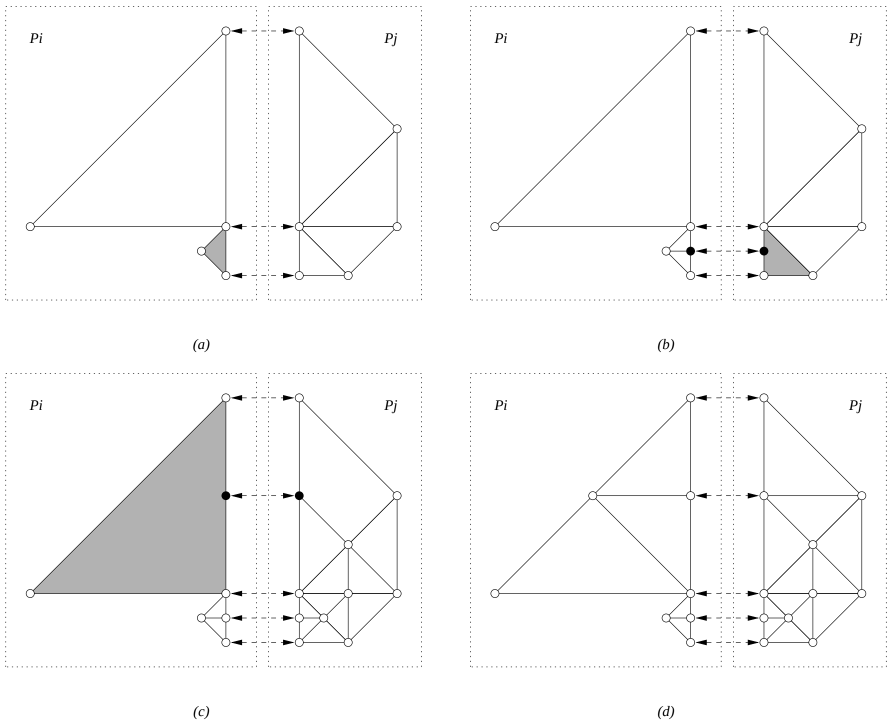


Figure 11: A cyclic propagation using the parallel longest edge bisection algorithm. (a) The refinement of an element (shaded) in P_i propagates to P_j in (b) and back to P_i in (c). When the procedure terminates the mesh is conforming as shown in (d).

Let R be a set of elements to be refined
while there is an element $\Omega_i \in R$ **do**
  bisect Ω_i by its longest edge
  add any non-conforming element Ω_j to R
end while

Figure 12: General longest-edge bisection (GLB) algorithm.

It is not immediately obvious if the resulting meshes obtained by the serial and parallel longest edge bisection algorithms are the same or if different partitions of the mesh generate the same refined mesh. As we mentioned earlier, messages can arrive from different sources in different orders and elements may be selected for refinement in different sequences.

We now show that the meshes that result from refining a set of elements R from a given mesh M using the serial and parallel algorithms described in Sections 4 and 5, respectively, are the same. In this proof we use the general longest-edge bisection (GLB) algorithm outlined in Figure 12 where the order in which elements are refined is not specified. In a parallel environment, this order depends on the partition of the mesh between processors. After showing that the resulting refined mesh is independent of the order in which the elements are refined using the serial GLB algorithm, we show that every possible distribution of elements between processors and every order of parallel refinement yields the same mesh as would be produced by the serial algorithm.

Theorem 6.1 *The mesh that results from the refinement of a selected set of elements R of a given mesh M using the GLB algorithm is independent of the order in which the elements are refined.*

Proof: An element Ω_i is refined using the GLB algorithm if it is in the initial set R or refinement propagates to it. An element $\Omega_i \notin R$ is refined if one of its neighbors creates a non-conforming vertex at the midpoint of one of its edges. The refinement of Ω_i by its longest edge divides the element into two nested subelements Ω_{i_1} and Ω_{i_2} called the children of Ω_i. These children are in turn refined by their longest edge if one of their edges is non-conforming. The refinement procedure creates a forest of trees of nested elements where the root of each tree is an element in the initial mesh M and the leaves are unrefined elements. For every element $\Omega_i \in M$, let τ_i be the refinement tree of nested elements rooted at Ω_i when the refinement procedure terminates.

Using the GLB procedure elements can be selected for refinement in different orders, creating possible different refinement histories. To show that this cannot happen we assume the converse, namely, that two refinement histories H_1 and H_2 generate different refined meshes, and establish a contradiction. An element $\Omega_i \in M$ such that the refinement trees τ_i^1 and τ_i^2, associated with each refinement histories H_1 and H_2 of Ω_i respectively, are different. Because the root of τ_i^1 and τ_i^2 is the same in both refinement histories, there is a place where both trees first differ. That is, starting at the root, there is an element Ω_j that is common to both trees but for some reason, its children are different. Because Ω_j is always bisected by the longest edge, the children of Ω_j are different only when Ω_j is refined in one refinement history and it is not refined in the other. In other words, in only one of the histories Ω_j has children.

Because Ω_j is refined in only one refinement history, then $\Omega_j \notin R$, the initial set of elements to refine. This implies that Ω_j must have been refined because one of its edges became non-conforming during one of the refinement histories. Let D_1 be the set of elements that are present in both refinement histories, but are refined in H_1 and not in H_2. We define D_2 in a similar way.

For each refinement history, every time an element is refined, it is assigned an increasing number. Select an element Ω_i from either D_1 or D_2 that has the lowest number. Assume that we choose Ω_i from D_1 so that Ω_i is refined in H_1 but not in H_2. In H_1, Ω_i is refined because a neighboring element Ω_j created a non-conforming vertex at the midpoint of their shared edge e. Therefore Ω_j is refined in H_1 but not in H_2 because otherwise it would cause Ω_i to be refined in both sequences. This implies that Ω_j is also in D_1 and has a lower refinement number than Ω_i contradicting the definition of Ω_i. It follows that the refinement histories are the same. ■

Consider now the parallel refinement algorithm described in Section 5. Although it refines elements on individual processors serially, these processors refine their elements in parallel. As shown below, the resultant mesh is the same as would be produced in the serial case.

Corollary 6.2 *Given a mesh M and a set of elements R to refine, for every partition of the elements between processors, the parallel GLB algorithm generates the same refined mesh as does the serial GLB algorithm.*

Proof: For every partition of the elements and every order of refinement within processors that is consistent with the GLB algorithm, there is a linear order that can be established of element refinements. From Theorem 6.1 it follows that the refined mesh associated with every linear ordering is the same. ■

Clearly the result of Theorem 6.1 holds for any nested refinement algorithm in which the refinement of elements occurs in an order that is independent of which side or face is non-conforming.

7 Experimental Results

In the previous section we have shown that the meshes obtained using the parallel refinement algorithm described in this paper are the same as those obtained using the widely used serial longest-edge bisection algorithm. Therefore the quality of the resulting meshes using any of the measures mentioned in Section 2 is the same as that obtained using serial refinement.

To evaluate the performance of our parallel algorithm and to show that refinement does not incur any significant overhead we performed a series of tests using a network of four to thirty-two Sun Ultra-1 workstations, each with 128Mb of memory, running Solaris 2.6 and connected through a 100Mbs network. In these tests we worked with meshes that contained millions of elements and vertices. Using simple regular meshes we first demonstrate that the performance of the parallel refinement algorithm strongly depends on the quality of the partition of the mesh between processors. If many adjacent elements are located in different processors the cost of communication is very high. Nevertheless, for good partitions of the mesh there is a reasonable communication overhead.

We also show the performance of the refinement algorithm on irregular unstructured meshes is similar to that observed on the simpler regular ones. Finally, we examine the performance of refinement algorithm when locally adapting unstructured meshes and compare it with the other phases of the adaptation procedure.

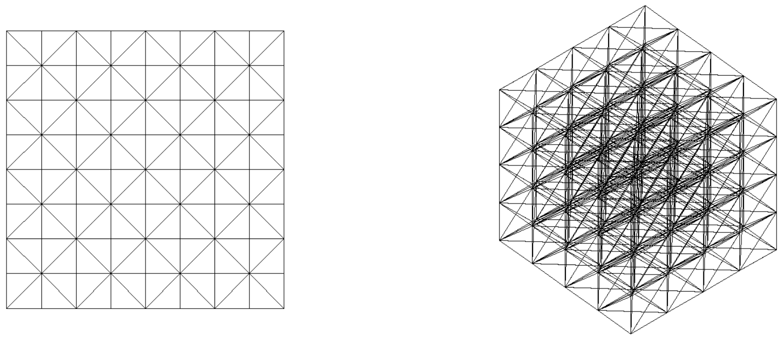


Figure 13: Initial two- and three-dimensional regular meshes.

7.1 Global Refinement of Regular Meshes

The performance of our parallel refinement algorithm is dependent on the quality of the partition of the mesh. Starting from the regular two- and three-dimensional meshes defined in the unit square and cube, as shown in Figure 13, we performed successive global parallel refinements of all the elements of the mesh. Each bisection of every element of these regular meshes doubles the number of elements in the mesh. In our two-dimensional example we start with a mesh that contains 256 elements and 145 vertices and after 15 successive global refinements we obtain a mesh with 4,194,304 elements and 2,099,201 vertices. The initial three-dimensional mesh contains 1,536 elements and 426 vertices. After 12 refinements its number of elements and vertices grows to 3,145,728 and 536,769, respectively.

PARED allows the user to choose from a large variety of partitioning algorithms. One of these is Multilevel-KL, a serial graph partitioning algorithm offered as part of the Chaco package [11]. It generates high quality partitions of meshes but at the cost of a large overhead that is prohibitively large for meshes. Multilevel-KL was used in the experiments described below to provide good partitions for the initial assignment of elements to processors. Because Multilevel-KL is a serial algorithm, an entire graph must be moved to one processor to partition it. For very large refined meshes, the time required by Multilevel-KL is very large (on the order of 20 minutes in some cases) and required the use of a Sun server with 2 GB of memory. We have developed an alternative incremental partitioning heuristic called Parallel Nested Repartitioning (PNR) [6, 12] that very quickly generates very high quality partitions incrementally as a mesh is refined and coarsened. A snapshot of the refined regular meshes partitioned randomly and by Multilevel-KL from a random partition are shown in Figure 14.

We used three different measures to evaluate the performance of the parallel refinement algorithm. For every processor, we count the number of new edge refinements sent to other processors during each of the successive global refinements of the mesh. This is a measure of the amount of communication occurring in the parallel refinement algorithm. The maximum number of edge refinements sent by any processor to all its neighbors in each level using both partitioning strategies are shown in Table 1 for the regular two-dimensional mesh and in Table 2 for the three-dimensional mesh. The amount of communication during refinement required for a random distribution of the mesh is significantly larger than for the mesh partitioned using Multilevel-KL. Also, the number of new shared edges using Multilevel-KL does not necessarily increase with each level of refinement even on a regular mesh because it depends on the actual partition of the mesh between proces-

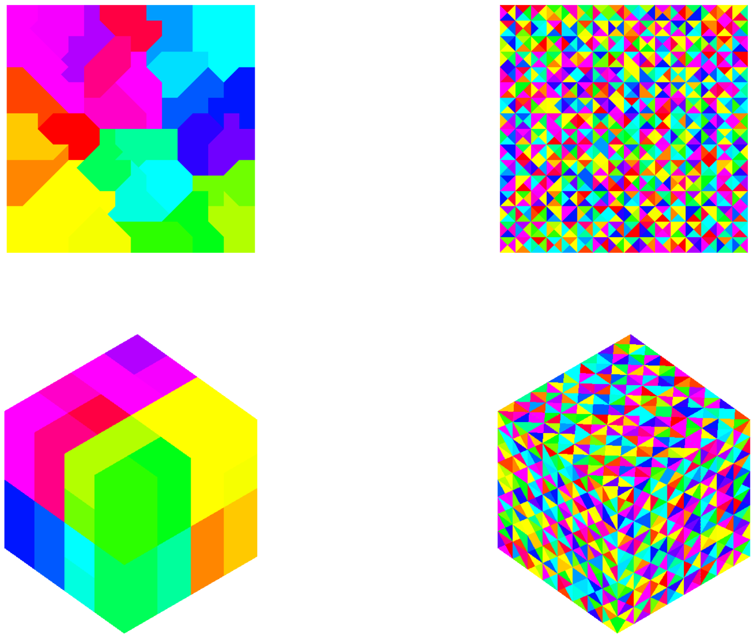


Figure 14: Regular two- and three-dimensional regular meshes partitioned between 32 processors using a partition provided by Multilevel-KL (left) and a random partition (right).

Level	4 Processors		8 Processors		16 Processors		32 Processors	
	M-KL	Rand	M-KL	Rand	M-KL	Rand	M-KL	Rand
1	6	64	6	39	6	20	4	12
2	4	120	6	90	5	43	6	26
3	0	237	0	170	3	89	3	55
4	13	504	11	320	9	167	9	84
5	4	1024	8	677	9	348	5	151
6	19	2075	24	1326	21	615	21	281
7	11	4129	10	2740	25	1233	13	504
8	43	8095	33	5314	32	2421	28	1003
9	25	16626	24	10800	30	4840	24	1964
10	66	32786	82	21307	70	9789	48	3910
11	48	66010	55	42844	83	19399	76	7899
12	99		158	85425	111	38291	100	15525
13			120		145	76806	89	30952
14					241		206	61452
15							207	

Table 1: Largest number of edge refinements sent by a processor to other processors in each refinement level obtained by performing successive refinements on a regular 2D mesh partitioned randomly and with the Multilevel-KL algorithm.

sors. Therefore, it is difficult to predict the amount of communication required by the refinement algorithm even in these simple examples.

These differences in the amount of communication also have a great impact on the refinement time as we increase the size of the mesh and the number of processors. For the regular two-dimensional mesh, the total time for each parallel refinement phase of random and Multilevel-KL derived partitions is shown in Figures 15 (a) and (b) respectively. For the three- dimensional mesh, these times are shown in Figures 15 (c) and (d). These figures show that the refinement time for randomly distributed meshes is much larger than that for the same meshes partitioned using Multilevel-KL. It is also possible to process larger meshes with Multilevel-KL because it requires fewer shared copies of vertices and less memory to store them.

The speedup in the time to refine a mesh using the GLB algorithm is shown in Figure 16 for random and Multilevel-KL-derived partitions of regular 2D and 3D meshes. We show only the time to globally refine all elements and propagate changes across processor boundaries. The time to repartition a random mesh with Multilevel-KL and to migrate the mesh are not included. In both the 2D and 3D cases the speedup that is obtained with the well partitioned mesh is much higher than for the randomly partitioned mesh and almost maximal. Also, we can report that the refinement time is dominated by the time for local refinement of meshes on individual processors, not the cost of interprocessor communication. Figure 16 (a) shows the speedup obtained by refining the regular two-dimensional mesh with 262,144 elements for both the random and Multilevel-KL derived partitions. Figure 16 (b) shows the speedup obtained by the refinement of the regular three-dimensional mesh with 98,304 elements. Due to the large number of new shared vertices created in randomized partitions, the communication cost overhead remains an important factor in the total cost of the refinement algorithm as the meshes grow in size.

	4 Processors		8 Processors		16 Processors		32 Processors	
Level	**M-KL**	**Rand**	**M-KL**	**Rand**	**M-KL**	**Rand**	**M-KL**	**Rand**
1	23	422	21	559	21	521	15	302
2	41	644	30	890	32	958	16	606
3	0	1273	0	1952	0	2044	0	1569
4	78	3230	80	4265	51	3800	24	2482
5	90	4731	87	7393	66	7800	46	5669
6	7	10199	16	15755	18	16997	4	12617
7	260	25778	229	34332	343	32578	191	20808
8	425		323	57350	302	65188	203	49028
9	141		125		98	139756	101	104314
10			1278		986		886	173428

Table 2: Largest number of edge refinements sent by a processor to other processors in each refinement level obtained by performing successive refinements on a regular 3D mesh partitioned randomly and with the Multilevel-KL algorithm.

7.2 Global Refinement of Irregular Meshes

The repeated bisection of all the elements in regular meshes never creates a mesh whose smallest angle is less than 45 degrees in the the case of two-dimensional meshes and a smallest solid angle less than 15 degrees in the the case of three-dimensional meshes. Also, it never refines an element more than once to maintain the conformality of the mesh.

Typical unstructured finite-element meshes are similar to those shown in Figure 17. Successive refinement of these meshes more than doubles the size of the mesh, as shown in Table 3. That is, an element might be refined more than once during the refinement phase, which is not the case in the previous regular examples.

Nevertheless, the performance of the algorithm using these irregular meshes is similar to our previous results. Table 4 shows that the maximum number of new edge refinements sent by any processor to another in each refinement is relatively small. The total refinement time in the different steps of the refinement are comparable to the ones in the regular meshes, as shown in Figure 18.

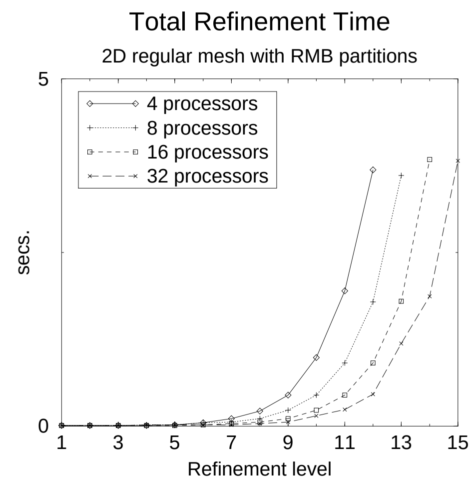
7.3 Local Adaptive Refinement of Irregular Meshes

We have mentioned earlier that local refinement of the mesh using the longest edge bisection algorithm can potentially cause refinement to propagate through all the elements of the mesh and, in the parallel case, through all the processors. In the previous examples, there is very little propagation because all the elements have similar size. By performing repeated global refinements of the mesh it is not likely that the refinement of an element would propagate to distant regions of the mesh requiring several messages between processors to obtain a conforming mesh.

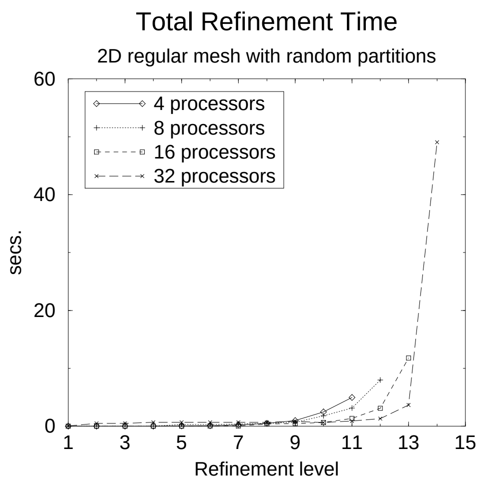
A good static test for our framework and one for which the best mesh is not regular is the two-dimensional problem defined by Laplace's equation $\Delta u = 0$ in the square $\Omega = (-1,1)^2$ with the following Dirichlet boundary conditions [26]:

$$g(x,y) = \cos(2\pi(x-y))\frac{\sinh(2\pi(x+y+2))}{\sinh(8\pi)}.$$

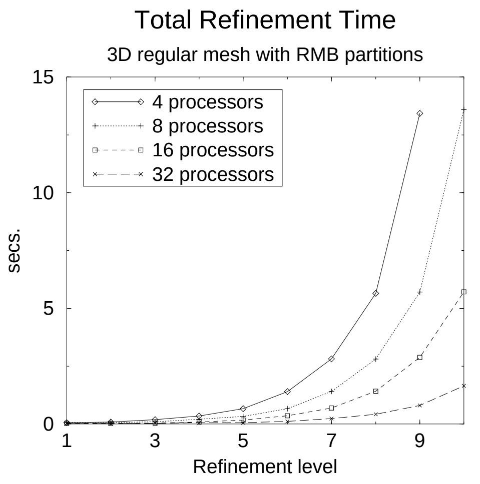
The analytical solution for this problem is known to be $u(x,y) = g(x,y)$ at every point of the domain Ω. This solution is smooth but changes rapidly close to the corner $(1,1)$. A similar problem has been



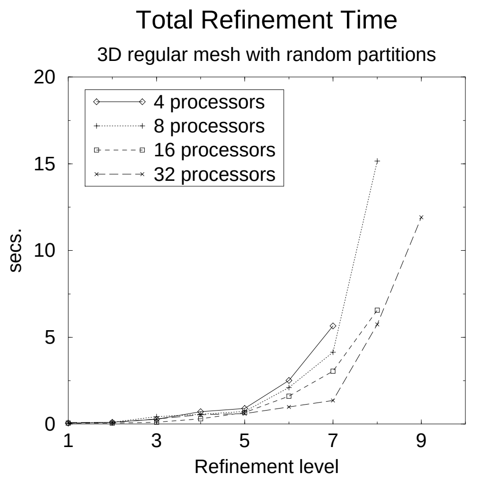
(a)



(b)



(c)



(d)

Figure 15: Refinement time for the global successive refinements of a regular two-dimensional mesh with a (a) Multilevel-KL partition and (b) random partition. (c) and (d) show the time for the refinement of a regular three-dimensional mesh.

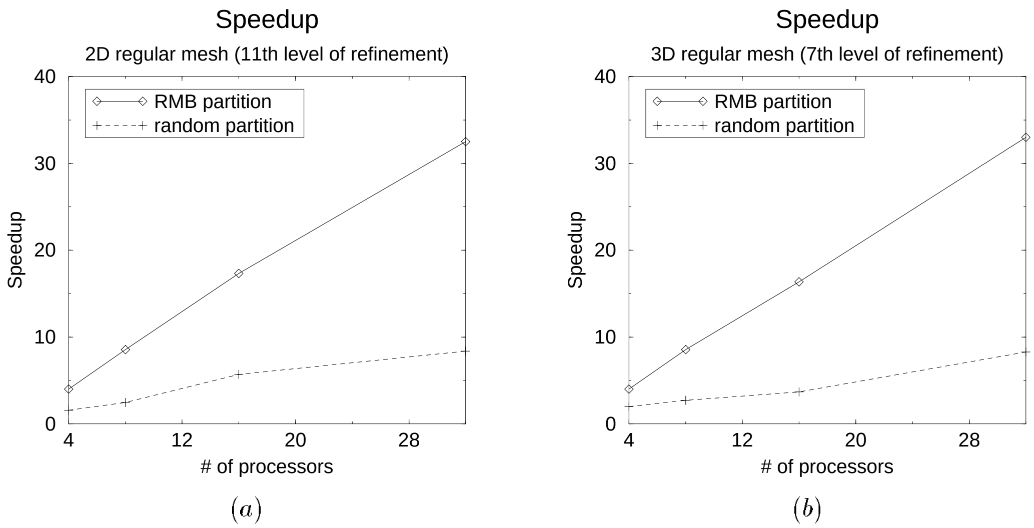


Figure 16: Relative speedups for the refinement of a regular mesh using Multilevel-KL and random partition for a (a) two-dimensional regular mesh of 262,144 elements and (b) three-dimensional regular mesh of 98,304 elements.

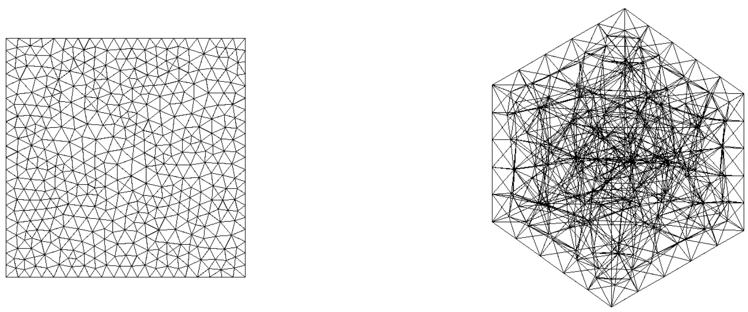


Figure 17: Initial two- and three-dimensional irregular meshes on a square and a cube.

Level	2D Irregular Mesh		3D Irregular Mesh	
	Elements	Vertices	Elements	Vertices
1	2239	1171	3179	728
2	5094	2629	10166	2177
3	11110	5677	31935	6472
4	23749	12040	101051	19554
5	49915	25214	320424	60762
6	103585	52125	1007549	187599
7	212831	106941	3177713	585128
8	434119	217726		
9	880745	441436		
10	1779869	891269		
11	3586501	1795390		

Table 3: Number of elements and vertices that result from successive refinement of irregular two- and three-dimensional meshes.

Level	2D Irregular Mesh				3D Irregular Mesh			
	4 Proc	8 Proc	16 Proc	32 Proc	4 Proc	8 Proc	16 Proc	32 Proc
1	21	22	17	15	51	52	51	48
2	14	22	16	14	146	109	92	79
3	19	31	17	23	213	162	187	153
4	26	29	31	20	433	383	336	307
5	31	36	36	33		811	744	613
6	37	47	35	41			1613	1221
7	87	56	65	62				
8	73	131	83	74				
9		129	143	101				
10			190	143				
11				206				

Table 4: Largest number of new edge refinements sent by a processor to other processors in each refinement phase by performing successive refinements on an irregular 2D and 3D meshes (Multilevel-KL partitions).

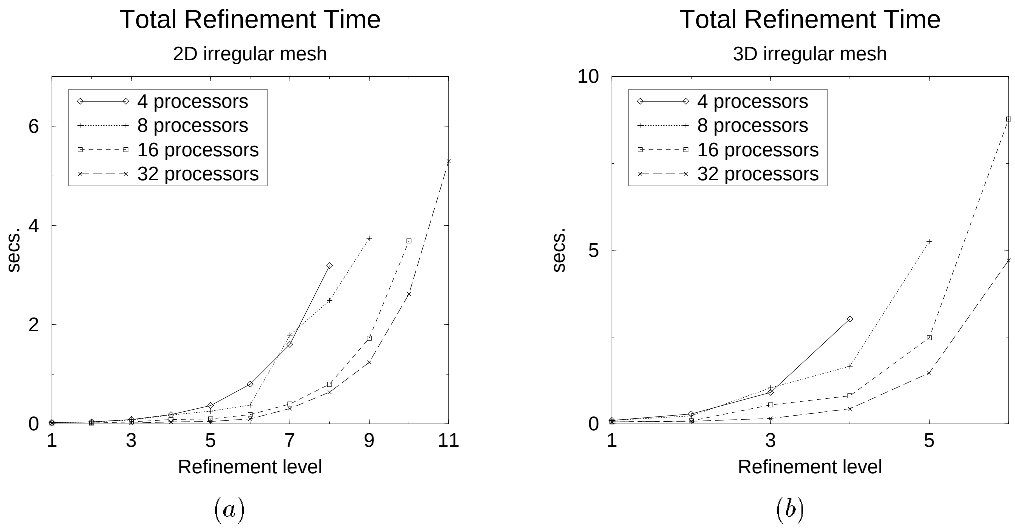


Figure 18: Refinement time for the global successive refinements of a irregular (a) two-dimensional mesh and (b) three-dimensional irregular mesh when Multilevel-KL is used.

defined in three dimensions. We use these two problems to study the performance of the refinement algorithm and compare it with the other components of PARED.

First we loaded into the coordinator the two- and three-dimensional meshes with elements of approximately uniform size, as shown in Figure 17. We computed a partition of these meshes using the Chaco [11] graph partitioning library and we distributed mesh elements between our workstations.

After setting a maximum error criterion the simulation is started in parallel. PARED solves the equations in parallel and computes an error estimate. While the estimate exceeds the maximum error desired, the mesh is adapted locally, the load balanced using PNR, work migrated, and the equations solved again using the new mesh. For these tests we solved the system of equations in parallel using the Conjugate Gradient method with Jacobi preconditioner. In this example of a static problem, only refinement was used. Each refinement phase increases the number of mesh elements and vertices in the region of rapid solution change, as shown in Figure 19. Using linear basis functions more than 1 million elements and 17 levels of local refinement were needed to achieve the desired error in the 2D case and about 800,000 and 11 levels of refinement were needed in the 3D case.

We examined the time spent by PARED in each of the solution, refinement, partition, and migration phases for the 2D and 3D versions of Laplace's equation described above. The results from these experiments are shown in Figures 20 and 21 for the 2D and 3D problems, respectively. There are several things that stand out from these experiments. First, in the 2D case the time spent solving the equations clearly dominates the time to refine elements and partition and migrate them. Second, in the 3D case the time spent solving equations is comparable to the time for refinement and migration while the time for partitioning is negligible.

The local refinement time varies widely between processors. When the mesh is partitioned between processors it is likely that most of the elements that are refined will be located on one or a few processors. As mentioned earlier, each element selected for refinement is bisected by its

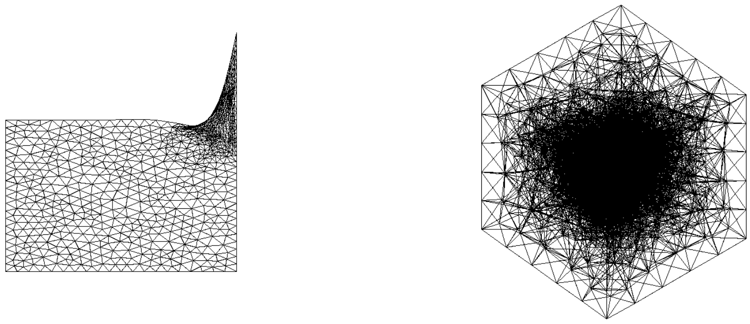


Figure 19: Irregular two- and three-dimensional regular meshes adaptively refined to solve Laplace's equation of a problem that exhibits high physical activity in one of its corners.

longest side and the refinement propagates to adjacent elements and across processor boundaries to maintain the conformality of the mesh. In this phase, the largest amount of time is spent refining elements that are local to the processor; communication overhead is small.

For every refinement element, we also computed the *processor depth* of the propagation, that is, how many messages between processors are required to make that element conformant. Even in these examples with very high dissimilar physical scales, the depth is never more than 4 so it is unlikely that the refinement will propagate through a large number of processors.

8 Related work

PARED follows in the spirit of the Distributed Irregular Mesh Environment (DIME) [27] by Roy Williams at the California Institute of Technology. DIME is an early system for the adaptive solution of PDEs using 2D unstructured meshes. DIME uses a database of vortices to maintain the identity of each mesh structure.

The Scalable Unstructured Mesh Computation (SUMAA3d) project at the Argonne National Laboratories is another related project, but for 2D problems. In SUMAA3d the mesh is partitioned by elements rather than vertices, although each element is owned by only one processor. To avoid the synchronization problem of refining adjacent elements in neighboring processors, SUMAA3d uses heuristics [4]. Each element is assigned a random color and elements with the same color are refined in the same phase. Because adjacent elements have a high probability of having a different color, they are not likely to be refined at the same time.

In [28] an adaptive environment for unstructured problems called PMDB is presented. PMDB refines tetrahedrons using a variety of different subdivision patterns. PMDB first marks a set edges for refinement. Then each element is refined into two or more tetrahedrons depending on which of its edges are marked. In the case that this procedure creates a tetrahedron with very sharp angles, then some adjacent tetrahedrons might be additionally refined but only if they are located in the same processor. PMDB does not propagate the refinement across processor boundaries.

As noted above, the PARED refinement procedure is general and suffers from none of the restrictions that apply to the refinement procedures of above three parallel systems for solve PDEs via the finite element method.

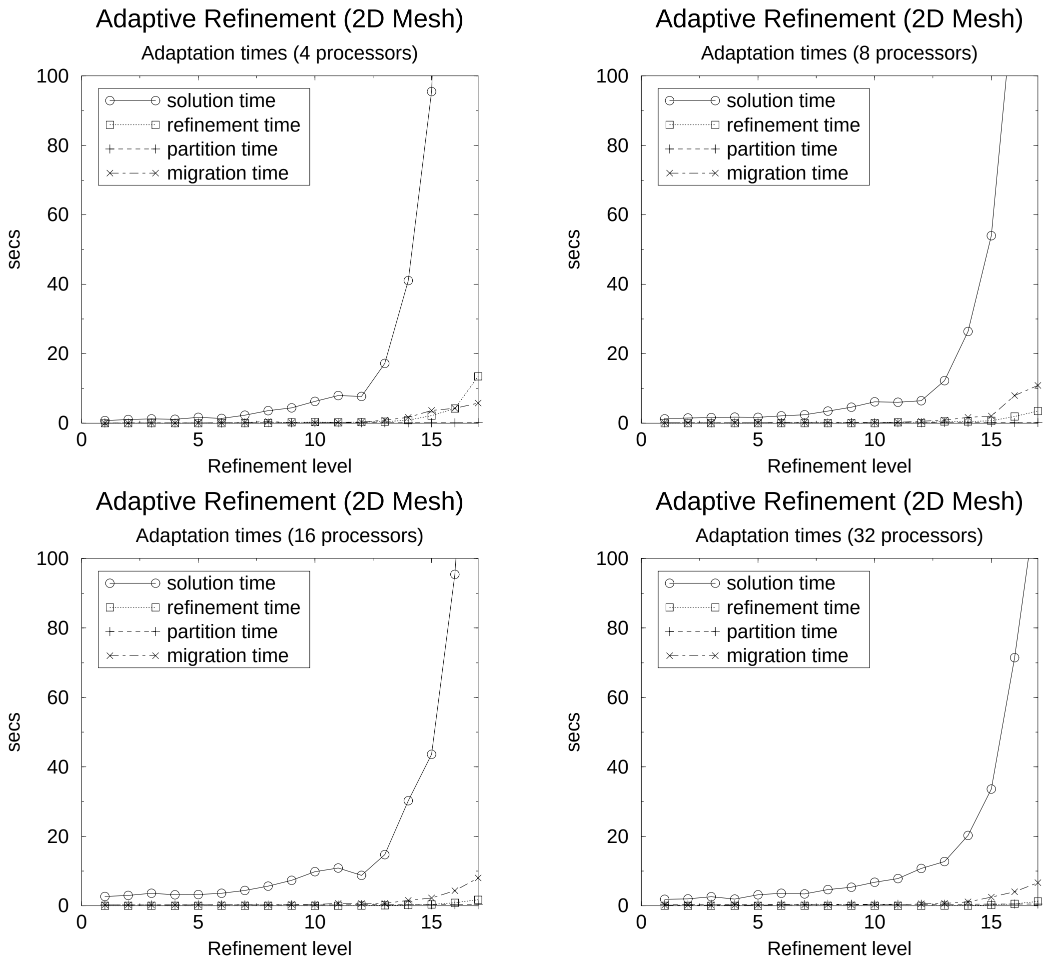


Figure 20: Adaptation times for two-dimensional mesh.

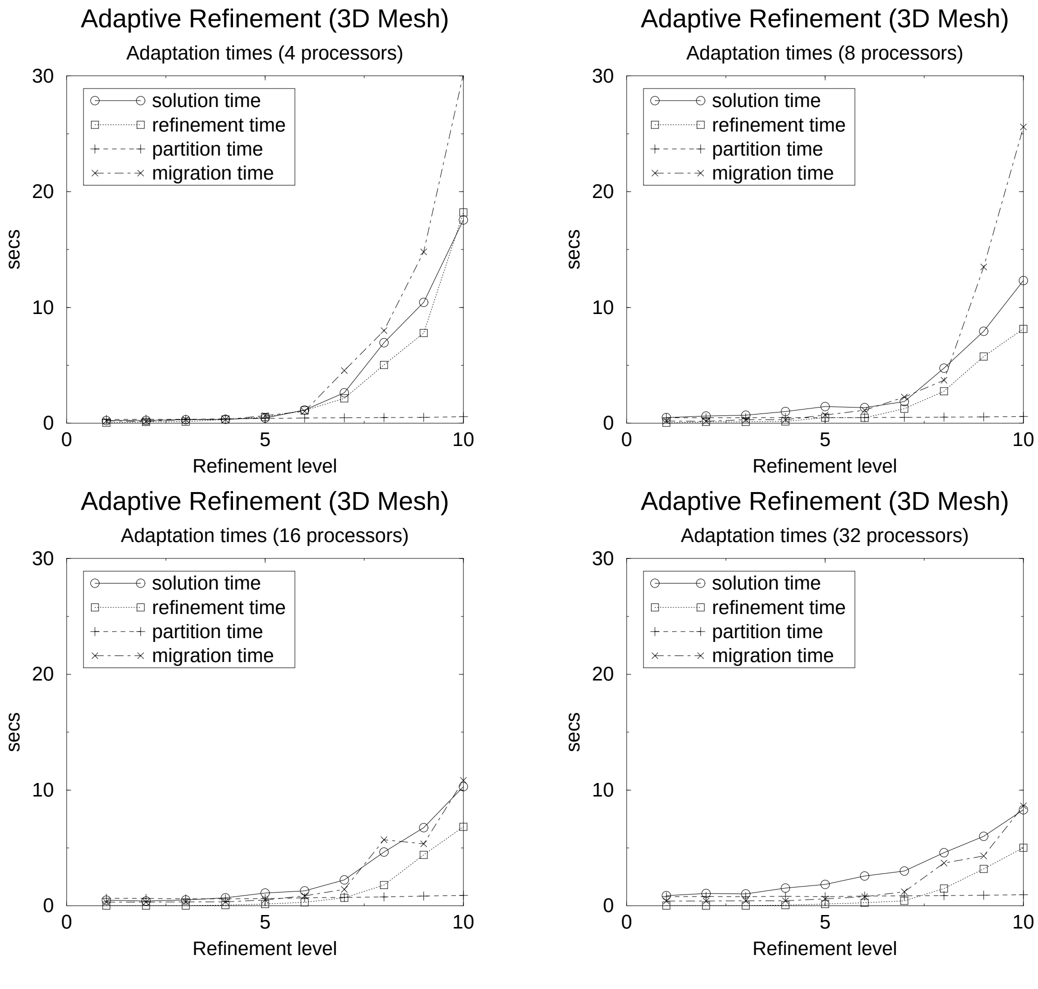


Figure 21: Adaptation times for three-dimensional mesh.

9 Conclusions

In this paper we have presented an algorithm for the refinement of two- and three-dimensional unstructured meshes that are distributed between multiple processors. We build on the work done for similar serial refinement algorithms by proving that the meshes generated by the parallel refinement algorithm are identical to those obtained by the serial ones.

We have shown that our parallel refinement algorithm is efficient and does not require significant overhead. Nevertheless, good partitions of the mesh are still required. In that case, we obtain nearly linear speedups when refining all the elements of the mesh.

We have also used the refinement algorithm to locally adapt the mesh on distributed memory machines. Speedup is not a meaningful measure to evaluate the performance of the refinement algorithm in this case because the refinement always occurs in one or few processors. Nevertheless, the local refinement times are comparable to the times spent in other phases of the adaptation procedure, such as solving and migration.

References

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

[2] Maria Cecilia Rivara. Selective refinement/derefinement algorithms for sequences of nested triangulations. *International Journal for Numerical Methods in Engineering*, 28:2889–2906, 1989.

[3] Maria Cecilia Rivara. A 3-D refinement algorithm suitable for adaptive and multi-grid techniques. *Communications in Applied Numerical Methods*, 8:281–290, 1992.

[4] Mark T. Jones and Paul E. Plassmann. Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. In *Proceedings of the Scalable High-Performance Computing Conference*, 1994.

[5] Jose G. Castanos and John E. Savage. The dynamic adaptation of parallel mesh-based computation. Technical Report CS-96-31, Department of Computer Science, Brown University, October 1996.

[6] Jose G. Castanos and John E. Savage. The dynamic adaptation of parallel mesh-based computation. In *SIAM 7th Symposium on Parallel and Scientific Computation*, 1997.

[7] I. Babuska and A. Aziz. On the angle condition in the Finite Element Method. *Int. J. Numer. Meth. Eng.*, 12, 1978.

[8] A. Liu and B. Joe. Relationship between tetrahedron shape measures. *BIT*, 34, 1994.

[9] V.N. Parthasarathy, C.M. Graichen, and A.F. Hathaway. A comparison of tetrahedron quality measures. *Finite Elements in Analysis and Design*, 15:255–261, 1993.

[10] Roy Williams. Adaptive parallel meshes with complex geometry. *Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, 1991.

[11] B. Hendrickson and R. Leland. The Chaco user's guide, version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, 1995.

[12] Jose G. Castanos and John E. Savage. Partitioning unstructured adaptive meshes. Technical Report in preparation, Department of Computer Science, Brown University, 1999.

[13] Message Passing Interface Forum:. MPI: A message passing interface standard, 1994.

[14] W. Gropp, E. Lusk, and A. Skellum. *Using MPI: Portable parallel programming with the Message Passing Interface*. MIT Press, 1994.

[15] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The complete reference*. MIT Press, 1996.

[16] *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.1*. Addison-Wesley, 1997.

[17] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[18] H. D. Simon. Partitioning of unstructured meshes for parallel processing. *Computing Systems Eng.*, 1991.

[19] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *Proceedings of the 6th SIAM conference on Parallel Processing for Scientific Computing*, pages 711–718, 1993.

[20] E. J. Schwabe, G. E. Blelloch, A. Feldmann, O. Ghattas, J. R. Gilbert, G. L. Miller, D. R. O'Hallaron, J. R. Shewchuck, and S. Teng. A separator-based framework for automated partitioning and mapping of parallel algorithms for numerical. In *Proc. 1992 DAGS Symposium*, 1992.

[21] M. Mamman and B. Larrouturou. Dynamical mesh adaptation for two-dimensional reactive flow simulations. In *Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*. North-Holland, Amsterdam, 1991.

[22] R. E. Bank, A. H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. *Scientific Computing*, 1983.

[23] I. G. Rosenberg and F. Stenger. A lower bound on the angle of triangles constructed by bisecting the longest side. *Mathematics of Computation*, 29(130):390–395, 1975.

[24] E. W. Dijkstra and C. S. Sholten. Termination detection for difussing computations. *Inf. Proc. Lett.*, 11, 1980.

[25] Dimitri P. Bertsekas and John N. Tsisiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, 1989.

[26] Ulrich Rüde. *Mathematical and Computational Techniques for Multilevel Adaptive Methods*. SIAM, 1993.

[27] R. D. Williams. DIME: A user's manual. Technical Report C3P 861, Caltech Concurrent Computation, 1990.

[28] M. S. Shephard, J. E. Flaherty, H. L. DeCougny, C. Ozturan, C. L. Bottasso, and M. Beall. Parallel automated adaptive procedures for unstructured meshes. In *Parallel Computing in CFD*. AGARD, 1995.

[29] C. Ozturan, H.L. deCougny, M. S. Shephard, and J. E. Flaherty. Parallel adaptive mesh refinement and redistribution on distributed memory computers. Technical Report TR93-26, Department of Computer Science, Rensselaer Polytechnic Institute, 1993.