The Dynamic Adaptation of Parallel Mesh-Based Computation

José G. Castaños and John E. Savage

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-96-31
October 1996

Abstract

We present efficient algorithms and data structures for dynamic refinement/coarsening (adaptation) of unstructured FE meshes on loosely coupled parallel processors. We describe a) a parallel adaptation algorithm, b) an online parallel repartitioning algorithm based on mesh adaptation histories, c) an algorithm for the migration of mesh elements between processors, and d) an integrated object-oriented framework for the adaptation, repartitioning and migration of the mesh at runtime. A two-dimensional triangle-based prototype demonstrates the feasibility of these ideas.

1 Introduction

Although massively parallel computers can deliver impressive peak performances, their computational power is not sufficient to simulate physical problems with highly localized phenomena by using only brute force computations. Adaptive computation offers the potential to provide large increases in performance for problems with dissimilar physical scales by focusing the available computing power on the regions where the solution changes rapidly. Since adaptivity significantly increases the complexity of algorithms and software, new design techniques based on object-oriented technology are needed to cope with the complexity that arises.

In this thesis we study problems that arise when finite-element and spectral methods are adapted to dynamically changing meshes. Adaptivity in this context means the local refinement and derefinement of meshes to better follow the physical anomalies. Adaptation produces load imbalances among processors thereby creating the need for repartitioning of the work load. We present new parallel adaptation, repartioning and rebalancing algorithms that are strongly coupled with the numerical simulation. Adaptation, repartitioning and rebalancing each offer challenging problems on their own. Rather than studying these problems individually we put special emphasis on investigating the way these different components interact. By considering adaptivity as a whole we obtain new results that are not available when these problems are studied separately.

We discuss the difficulties of designing parallel refinement algorithms and we introduce a refinement algorithm based on the Rivara's bisection algorithm for triangular elements [1], [2]. By representing the adapted mesh as a forest of trees of elements we avoid the synchronization problems for which Jones et al use randomization [3].

We propose a new Parallel Nested Repartitioning algorithm that has its roots in the multilevel bisection algorithm of Barnard et al [16]. It produces high quality partitions at a low cost, a very important requirement for recomputing partitions at runtime. It has a very natural parallel implementation that allows us to partition meshes of arbitrary size. The collapsing of the vertices is performed locally using the refinement history and avoiding the communication overhead of other partitioning methods [19]. Compared to iterative local migration techniques [42] this method does not require several iterations to rebalance the work.

Finally we design a mesh data structure where the elements and nodes are not assigned to a fixed processor throughout the computation but can easily migrate from one processor to another in order to rebalance the work. The mesh is represented as a set of C++ objects. To avoid the problem of having dangling pointers between different address spaces, the references to remote objects are handled through local proxies. These proxies keep track of the migration of objects as a result of load balancing.

To evaluate these ideas we designed and implemented a system in C++. This program runs on a network of workstations (NOW) and uses MPI [23] to communicate between processors. The most salient characteristic of adaptive codes is the high sophistication of their data structures. The use of object oriented techniques allowed us to reduce the complexity of the implementation without significantly affecting the performance.

2 Mesh-based computation

The numerical solution of complex partial differential equations using computational resources requires the definition of a domain Ω in which the problem is to be solved and a set of conditions to be applied at its boundaries [10]. The continuous domain and boundary conditions are discretized so they become amenable to computer manipulation. A computational mesh M is thereby produced. This mesh is constructed by splitting the domain into a set of simple polygons such as triangles and quadrilaterals (in 2 dimensions) or tetrahedrals (in 3 dimensions) called *elements* that are connected by *faces*, *edges* and *nodes*.

Once a mesh is constructed, elements can be split into a set of nested smaller elements or combined into a macroelement. This process is called the *adaptation* of the mesh. In an adaptive method the selective and local refinement of the mesh is interleaved with the solution of the problem by contrast with the static grid generation approach in which a fixed discretization of the geometry is done in a preprocessing step.

Adaptive methods can be schematically described as a feedback process where the automatic construction of a quasi-optimal mesh is performed in the course of the computation [1]. Rather than using a uniform mesh with grid points evenly spaced on a domain, adaptive mesh refinement techniques place more grid points where the solution is changing rapidly. The mesh is adaptively refined during the computation according to local error estimates on the domain [3]. Meshes are usually refined for two main reasons: [10]:

- to obtain a better solution by increasing the resolution in a particular region (*steady* case).
- to better resolve transient phenomena like shocks in the simulation of stiff *unsteady* two-dimensional flows [6]. During the computation the mesh is refined and coarsened (called sometimes fission and fusion operations) as

the regions of interest evolve and move. The construction of meshes for this type of problem requires data structures that allow:

- addition of elements when an element is refined by replacing it by two or more nested elements.
- coalescence of elements into larger elements when the mesh is coarsened.

Although the computational power of parallel computers is continuously increasing it is unlikely that they will reach the level of performance required to solve problems of very localized physical phenomena using a uniform discretization of the domain. Rather than using this brute force approach adaptive meshes restrict the use of small elements to the regions of interest while maintaining a coarser mesh everywhere else.

The use of adaptive meshes has the potential of producing large computational savings but at the price of significantly increasing the sophistication of codes and algorithms. As the mesh is no longer regular we need to develop new data structures that are usually more difficult to implement than the regular ones. Also the design of adaptive meshes in a parallel environment requires a close interaction between the algorithms that refine, partition and rebalance the mesh and the numerical simulation. The success of an adaptive strategy will depend strongly on how well these different modules can communicate with each other.

There is a wide variety of strategies for mesh refinement [8]. In the remaining part of this section we review some of the most common techniques for mesh generation and refinement. In the following section we introduce a strategy to implement adaptive meshes using a sequence of nested refinements. Later we show how to implement this approach on a parallel computer. We also explain the object-oriented techniques that we use to simplify the software design.

2.1 Selection of the mesh type

The selection of the mesh type depends on the problem to be studied since there is no strategy that it is considered best for every problem. Among the most common approaches we mention [8]:

- structured meshes: there is a mapping from the physical space to the computational space. In the computational space the elements appear as squares (in two dimensions) or cubes (in three dimensions) and the neighbors and vertices of an element are easily calculated using an array based data structure. The data structures for this type of mesh are very regular.

- unstructured meshes: in this case the elements store explicit connectivity information to determine their neighbors and vertices. The data structures in this case are more complex than in structured meshes but it is easier to represent complex geometries.

Each type of mesh has its advantages and disadvantages. Structured meshes require simpler codes with less overhead but are more limited in the representation of complex domains. Unstructured meshes are more complex, require more storage and overhead per element but can easily represent complex geometries and moving bodies. Some techniques implement the meshes as a combination of both approaches. In such cases the mesh is usually decomposed in a set of unstructured super-elements where each super-element is decomposed into a structured grid.

The choice of the mesh type determines the data structures and algorithms available for refinement, partitioning and rebalancing. For example, a partitioning method adequate for unstructured meshes such as Recursive Spectral Bisection [15] is useless for structured meshes. A refinement algorithm will perform well on some type of meshes but is not recommended for anothers. And the migration algorithm described in Section 7.2 highly depends on how the mesh is actually stored. In the rest of this paper we assume that the domain is discretized using unstructured meshes.

2.2 Mesh generation

The generation of meshes for unsteady problems is usually done in two distinct phases [10]:

- initial mesh creation: involves the creation of a compatible unstructured mesh using the geometry description of the problem domain. The complex topology of the problem is discretized into a set of simpler elements. This is a global process usually performed on a sequential computer and it might require human assistance.
- mesh adaptation: the selective refinement/coarsening of sections of the mesh improves the quality of the solutions either by increasing the resolution in interesting areas or by decreasing it on regions of little interest. The refinement of elements is largely a local process.

The compatibility of the mesh to the problem topology and correct treatment of the boundaries are not the only requirements for high-quality meshes. In addition it is desirable to have meshes whose elements are [1]:

- conforming: the intersection of elements is either a common vertex or a common side.
- non-degenerate: the interior angles of the elements are bounded away from zero.
- smooth: the transition between small and big elements is not abrupt.

2.3 Mesh adaptation

The following are the two principal strategies for mesh refinement [12] :

- h-refinement: is performed by splitting an element into two or more smaller subelements (refinement) or by combining two or more subelements into one element (coarsening). h is a parameter of the size of the elements. This method involves the modification of the graph structure of the mesh.
- p-refinement: can be thought as increasing the amount of information associated with a node without changing the geometry of the mesh [10], where p is the polynomial order of some element.

Through the rest of this paper we concentrate mainly on h-refinement although some of the techniques for mesh partitioning and migration are independent of the refinement strategy. Since p-refinement also modifies the workload in each processor the repartitioning and migration algorithms apply to it also.

2.3.1 Local h-refinement algorithms

Starting from a conforming mesh M formed by a set E of non-overlapping elements $E_i \in E$ that discretize a domain Ω of interest and a set of elements R, $R \subseteq E$, that are selected for refinement, h-refinement algorithms construct a new conforming mesh M' of embedded elements E'_i such that:

- if $E_i \in R$, E_i is split into a set of nonoverlapping subelements $\{E'_{i_1}, E'_{i_2}, \ldots, E'_{i_n}\}$ that replace E_i.
- if $E_i \notin R$, $E_i = E'_i$.

The selection of elements for refinement (or coarsening) in R is made by examining the values of an "adaptation criteria" [6] that can be related to a discretization error. Usually these refinement methods cause the *propagation* of the refinement to other mesh elements so an element $E_i \notin R$ might also be refined in order to obtain a conforming mesh. Coarsening algorithms have similar problems.

One common refinement algorithm is the Rivara bisection refinement algorithm for triangular elements that it is used in two dimensional problems. In its simplest form it bisects the longest edge of a triangle to form two new triangles with equal area. There are several variants of the serial bisection refinement algorithm. In Figure 1 we illustrate an example of the 2-triangles bisection algorithm [1] and [2] described in Figure 2. In Figure 1 (a) the element selected for refinement is shaded. The refinement of this element creates a non-conforming white node on its longest edge. The shaded element in 1 (b) must now be refined to to maintain a conforming mesh. This process is repeated in (c) where there are two non-conforming nodes. Finally in (f) we show the resulting mesh.

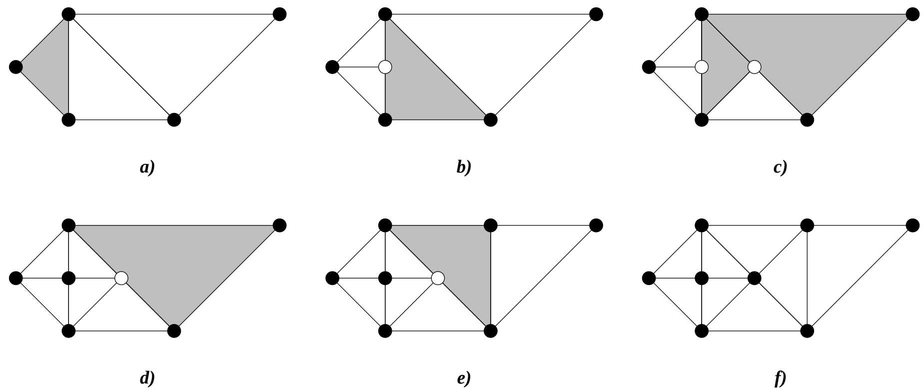


Figure 1: Bisection refinement: in (a) only one element is selected for refinement. (b) shows the mesh after the refinement of that element. A non-conforming white node is created so we propagate the refinement to an adjacent element. (c), (d) and (e) show different stages of the refinement and (f) shows the final mesh. Although only one element was initially in R we refined 3 more elements to obtain a conforming mesh.

Using the bisection refinement algorithm the propagation is guaranteed to terminate. Also if we start the refinement with a mesh M that has elements that are smooth, conforming and non-degenerate then the elements of the resulting mesh M' will also have the same properties.

3 Multilevel mesh adaptation

To support dynamic adaptation of meshes we designed a data structure based on a multilevel finite-element mesh with a filiation hierarchy between two consecutive levels. As we will show in later sections, our algorithms for refinement, partition and migration take good advantage of this mesh representation.

We assume that the user supplies an initial coarse mesh $M_0(E, V)$ called a *0-level* mesh where E is a set of elements and V is a set of nodes. This is the coarsest mesh that the refinement algorithm is able to manipulate. Using defined *adaptation criteria* we select some elements $E_i \in R \subseteq E$ for refinement and others $E_j \in C \subseteq E$ for coarsening.

For each refined element E_i we define the $Children(E_i) = \{E_{i_1}, E_{i_2}, \ldots, E_{i_n}\}$ to be the elements into which E_i is refined and let $Parent(E_{i_k}) = E_i$. Also for each element $E_i \in E$ we define $Level(E_i) = 0$ if E_i is in M_0 and $Level(E_i) = Level(Parent(E_i)) + 1$ otherwise. The children of an element E_i of level l can be further refined and they become the parents of level $l+2$ children. For each node V_p we define $Level(V_p) = 0$ if V_p is in M_0 and $Level(V_p) = Level(E_i) + 1$ if V_p was created as the result of refining an element E_i. Figure 3 gives an example of the refinement of a mesh along with the associated levels. Note that the meshes

FOR each $E_i \in R$ **DO**
bisect E_i by the midpoint of its longest side generating a new node V_p and two triangles E_{i_1} and E_{i_2}.
WHILE V_p is a non-conforming node in the side of some triangle E_j **DO**
make E_j conforming by bisecting E_j by its longest side generating a node V_q and two triangles E_{j_1} and E_{j_2}.
IF $V_p \neq V_q$ **THEN**
V_p is a non-conforming node in the midpoint of one of the sides of either E_{j_1} or E_{j_2}. Assume that V_p is in one side of E_{j_1}. Bisect E_{j_1} over the side that contains V_p obtaining two triangles E'_{j_1} or E''_{j_1}. Now V_p is a vertex of both triangles.
set $V_p = V_q$.
END IF
END WHILE
END FOR

Figure 2: Rivara's two-triangle refinement algorithm.

M_1, M_2 and M_3 in (b), (c) and (d) do not only include nodes and elements of the corresponding level but can also include nodes and elements of previous levels. Also the elements are replaced by their children when they are refined but the nodes are not. For example, every node V_p such that $Level(V_p) = 0$ will be present in all the successive meshes. Also note that some elements E_i of level l have as vertices nodes of level $l-1$ or less.

The iterative execution of this algorithm produces nested meshes. If M_0 is the coarsest mesh then for any level l:

$$M_l \prec M_{l-1} \prec \cdots \prec M_0$$

where $M_i \prec M_{i-1}$ is a relation that indicates that M_i has all the nodes present in M_{i-1} and that some elements in M_{i-1} have been split to form the elements in M_i.

3.1 Multilevel refinement

A sequence of nested refinements creates an element hierarchy. In this hierarchy each element of the initial mesh belongs to the *coarse mesh* M_0 and time $t > 0$ each element that it is not refined belongs to the *fine mesh*.

A decision to perform an n-fold refinement of $E_i \in R$ is transmitted to the refinement module as the pair (E_i, n). For example if $n = 1$ then using Rivara's bisection refinement the element E_i is divided into 2 triangles. If $n > 1$ then each of its children is refined $n-1$ times.

The multilevel algorithm for refinement has the following properties:

- an element that has no parents has level 0 and belongs to the *coarse (initial) mesh* M_0. No coarsening is done above this level.

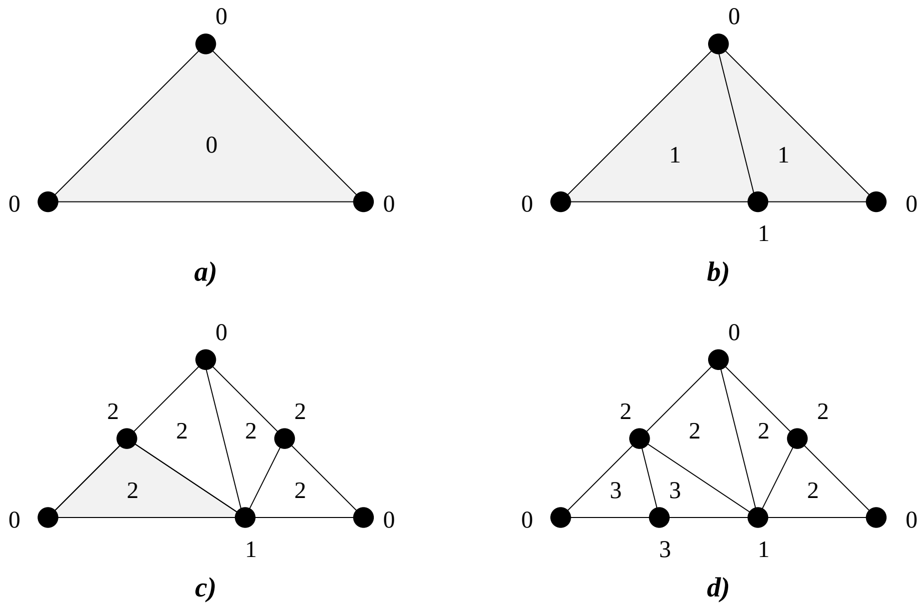


Figure 3: Refinement of a mesh. (a) shows the initial mesh M_0 while (b), (c) and (d) show the meshes M_1, M_2 and M_3. Associated with each node and element is its level.

- an element with no children belongs the *fine mesh* M_t. The numerical simulations are always based on the fine mesh.

- an element could be at the same time in both the *coarse mesh* M_0 and the *fine mesh* M_t (for example before any refinement is done) or in any intermediate mesh.

- only elements that are in the *fine mesh* M_t can be selected for refinement or coarsening. The hierarchy of elements is only modified at its leaves.

- a node V_p such that $Level(V_p) = l$ is a vertex of elements E_i of level l or below. An element E_i of level l has vertices of level m where $m \leq l$.

- as the elements are individually selected for refinement or coarsening the hierarchy can have different depths in different regions of the mesh.

- when an element E_i is refined it is replaced by its *children* in the *fine mesh* M_t. To coarsen an element all its *children* must be selected for coarsening. In this case the *children* in the *fine mesh* M_t are replaced by their *parent* and destroyed.

- both refinement and coarsening can propagate to adjacent elements. The algorithms are not completely local because they need to preserve conformality requirements.

This sequence of refinements is explained in Figure 4. Initially the elements E_a and E_b are selected for refinement (a). Under the mesh we show the internal representation. Both E_a and E_b belong to M_0 and M_t. After the first refinement 4 new elements are created. At this point M_t includes E_{a_1}, E_{a_2}, E_{b_1} and E_{b_2} (b).

3.2 Local coarsening

The selection of elements for coarsening is performed by evaluating an adaptation criterion at their vertices. The nodes in a finite element mesh are associated with the degrees of freedom and the unknowns of a system of equations. After finding its solution at time t the system might desire the elimination of nodes considered unnecessary according to a selected evaluation criterion to reduce the number of unknowns. This destruction of nodes requires coarsening of elements to maintain a conforming mesh.

The successful evaluation of the adaptation criteria at a node V_p is not a sufficient condition for the destruction of a node. Nodes created as a result of the propagation of refinement need to be adequately coarsened to preserve the conformality of the mesh. Elements at a lower level that reference the node need to be eliminated to prevent dangling references and this might require the destruction of other nodes. The conditions for a correct coarsening algorithm are:

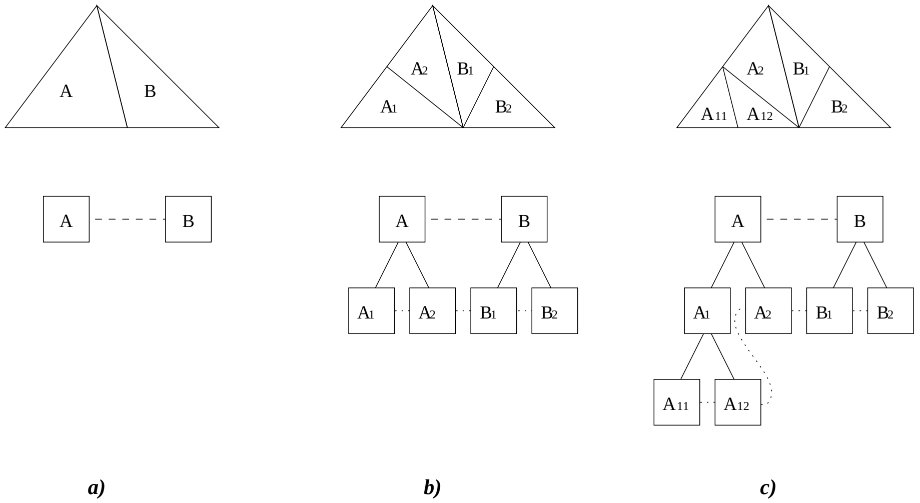


Figure 4: Multilevel refinement. Initially elements E_a and E_b in (a) are selected for refinement. Both elements are refined and replaced by their children (b). In (c) the element E_{a_1} is further refined. Under the meshes we show the corresponding mesh hierarchy.

- to select an element E_i of level l as a *candidate* for coarsening, all its vertices V_p that are nodes of level l should be selected as *candidates* for coarsening by evaluating the adaptivity criteria at the node.
- assume that this element E_i of level l is the child of some other element E_j of level $l-1$. In order to replace all the children of E_j by E_j all its children should be selected as a *candidates* for coarsening or none of them are.
- a node V_p is selected for coarsening, not anymore as a simple *candidate*, only if all its adjacent elements E_i are selected as *candidates* for coarsening. This condition prevents dangling references from elements to destroyed nodes.
- if an element E_i that is an element of level l has more than one vertex of level l and not all of them are selected for coarsening, then none of its vertices of level l is selected for derefinement since an element that has vertices of its level that are not selected for coarsening will not be coarsened and we need to prevent that its vertices allow the coarsening of adjacent elements.
- finally, an element E_i of level l is selected for coarsening if:
  - the element E_i has no children (it belongs to the fine mesh).

- the element E_i has a parent (it does not belong to the coarse mesh).
- its vertices are nodes of level m where $m \leq l$.
- all its vertices that are nodes of level l are selected for coarsening (and are not simple *candidates*). This last condition will ensure that the resulting mesh is conforming because a node V_p is selected for coarsening only if there will be no references to it.

4 The challenge of exploiting parallelism

The data structures and algorithms introduced in the previous section allow us to refine and coarsen a mesh in a serial computer. Most of the work that we will present in the rest of this paper extends these ideas to a parallel computer. Parallelism introduces a series of problems that we need to solve in order to perform the dynamic adaptation of parallel mesh-based computation.

Refinement algorithms typically use a local information to perform refinement. Unfortunately the refinement of an element E_a that creates a new node V_p in an internal boundary between two processors requires synchronization between the processors.

The second problem concerns with the termination of the refinement phase. The serial algorithm terminates when no more elements are marked for refinement. This is not always easy to detect in a parallel environment. In this case, global refinement termination holds only when all the processors have refined their elements and there is no propagation message in the network. A processor P_i might have no more local elements to refine but it needs to wait for possible propagations from neighbor processors. Only when all the processors agree on the termination of the refinement phase can they proceed to the next phase.

The adaptation of the mesh produces imbalances on the work assigned to each processor as elements and nodes are dynamically created and destroyed. Also mesh partitions are computed at runtime interleaved with the numerical simulation. In this environment we cannot afford expensive algorithms that recompute the partitions from scratch after each refinement. Instead we propose repartitioning algorithms that use the information available from previous partitions and the refinement history.

Finally we must keep a consistent mesh while migrating elements and nodes between processors. In our meshes the physical location of nodes and elements is not fixed throughout the computation. Instead our design supports dynamically changing connectivity information where the references to remote elements and nodes are updated as new nodes or elements are created, deleted or moved to a new processor to balance the work load.

In the following sections we address these problems in detail and we present our solutions. First however, we introduce some definitions, explain a strategy for storing meshes in a distributed memory parallel computer (that we call a *parallel mesh*) and show how to use the mesh to solve dynamic problems.

5 Mesh representation in a parallel computer

In Section 3 we presented a data structure to represent a refined mesh in a serial computer and we introduced serial refinement and coarsening algorithms. In this section we extend this data structure to store adapted meshes in parallel computers.

Let $M(E, V)$ be a finite element mesh where E is a set of elements and V is a set of nodes. We define $Adj(E_a) = \{V_p : V_p \text{ is a vertex of } E_a\}$. In a similar way we define $ElemAdj(V_p) = \{E_a : V_p \text{ is a node of } E_a\}$ and $NodeAdj(V_p) = \{V_q : V_p$ and V_q are both nodes of a common element $E_a\}$. $Adj(E_a)$ of an element E_a is the set formed by the vertices of E_a.

In the case of triangular elements $|Adj(E_a)| = 3$, and in the case of quadrilateral elements $|Adj(E_b)| = 4$. $ElemAdj(V_p)$ of a node V_p is the set formed by the elements adjacent to V_p and $NodeAdj(V_p)$ is the set formed by the nodes adjacent to V_p. Two nodes are considered adjacent not only because there is an edge between them in the mesh M but also if they are adjacent to a common element. In the case of quadrilateral elements two nodes at opposite corners are node adjacent. In an unstructured mesh $|NodeAdj(V_p)|$ is not a constant. Although in theory we can construct meshes where $|NodeAdj(V_p)|$ can have arbitrary values, if the mesh is non-degenerate (the interior angles are not close to 0) we expect that $|NodeAdj(V_p)|$ be close to a constant k.

A graph G is constructed from the finite element mesh M. Its adjacency matrix H has one row and column for each node $V_p \in V$. The entry $h_{p,q} = 1$ if the nodes V_p and V_q are adjacent to a common element and $h_{p,q} = 0$ otherwise. The adjacency matrix H can be directly constructed from $NodeAdj(V_p)$. Since $V_p \in NodeAdj(V_q) \Rightarrow V_q \in NodeAdj(V_p)$, the matrix H is symmetric and G is an undirected graph. In general $|NodeAdj(V_p)| \ll |V|$ so we expect that H will be very sparse.

5.1 Partitioning by elements or partitioning by nodes

In an iterative method for solving systems of equations the cost of the algorithm is dominated by the cost of performing repeated sparse matrix-vector products $Ab = c$ where A is $|V| \times |V|$. A and H have the same sparsity structure. This implies that a good partition for G is also a good partition for A because it minimizes the communication required to perform the matrix-vector products. There are two basic strategies for partitioning the graph G:

- *node-partitioning*: there is a partition $\Phi = \{\Phi_1, \Phi_2, \ldots, \Phi_P\}$ of the nodes between P processors such that $\bigcup \Phi_i = V$ and $\Phi_i \cap \Phi_j = \varnothing$, $\forall\, i \neq j$. If $V_p \in \Phi_i$ it is assigned to P_i. Each node is assigned to a single processor. The partition of G is performed by removing some edges, leaving sets of connected nodes. The edges removed express the communication pattern between processors and the cost of the partition is measured by the number of edges removed.

- *element-partitioning*: in this case there is a partition $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_P\}$ of the elements between P processors such that $\bigcup \Pi_i = E$ and $\Pi_i \cap \Pi_j = \varnothing,\ \forall\ i \neq j$. If $E_a \in \Pi_i$ it is assigned to P_i. Each element is assigned to a single processor. The partition is performed across the edges that separate two elements. If $V_p \in Adj(E_a)$ and also $V_p \in Adj(E_b)$ where $E_a \in \Pi_i$, $E_b \in \Pi_j$ and $i \neq j$ then V_p is a *shared node*. Both P_i and P_j have their own copy of V_p, that we will denote V_p^i and V_p^j respectively. Communication is required between multiple copies of the same node so the cost of the partition is measured by the number of shared nodes. We define $Nodes(\Pi_i) = \{V_p^i : E_a \in \Pi_i \text{ and } V_p \in Adj(E_a)\}$ hence $Nodes(\Pi_i)$ is the set of nodes corresponding to the elements in Π_i. Note that $Nodes(\Pi_i) \cap Nodes(\Pi_j) \neq \varnothing$ if the two partitions Π_i and Π_j are adjacent. To find a partition of the mesh using element-partitioning we first compute the dual $M^{-1}(E, W)$ of the mesh M where $W = \{(E_a, E_b) : E_a, E_b \in E, E_a \neq E_b, Adj(E_a) \cap Adj(E_b) \neq \varnothing\}$. W is a set of pairs of adjacent elements so they have at least one node in common. We then use a graph partitioning algorithm to assign elements to processors.

It is shown in [13] that partitioning by elements has several advantages over partitioning by nodes due to the way the matrix A is computed in the finite element method. The matrix A is the result of an *assembly* process. We first compute a local square matrix of $L(E_a)$ (of size $|Adj(E_a)|$) for each element $E_a \in E$. $L(E_a)$ represents the contribution of E_a to its nodes V_p. The global matrix A is equal to $\sum_{E_a \in E} L(E_a)$ (where $\sum$ means the direct sum of the local matrices after converting from the local indices in L to the global indices in A). The matrix A is also partitioned between the processors. If the node V_p is a shared node between two or more processors P_i and P_j then the entry in A_i corresponding to V_p^i has the contributions of the elements $E_a \in \Pi_i$ and the entry in A_j corresponding to V_p^j has the contributions of the elements $E_b \in \Pi_j$. The matrix A_i in processor P_i is *partially assembled* since it only considers the contributions of the elements $E_a \in P_i$. The *fully assembled* matrix is $A = \sum A_i$.

The matrix-vector product $Ab = c$ is performed in two phases. In the first phase each processor computes $A_i b = c_i$. The resulting vectors c_i are also partially assembled. In the second phase we communicate the individual vectors c_i to obtain $c = \sum c_i$.

5.2 Implementing a parallel mesh using remote references

A remote reference is a pair (P_i, V_p^i) where P_i is a processor and $V_p^i \in Nodes(\Pi_i)$. It represents a reference to the V_p^i copy of node V_p in processor P_i. We define $Ref(V_p^i) = \{(P_j, V_p^j) : V_p^j \text{ is a copy of } V_p \text{ in } P_j, i \neq j\}$. This relation is also symmetric so that if $(P_j, V_p^j) \in Ref(V_p^i)$ then $(P_i, V_p^i) \in Ref(V_p^j)$. The remote references are pointers to a remote address space. Since this is not allowed in almost any programming language we designed the remote references as C++ objects using the notion of *smart pointers*. We will come back to this when we discuss the implementation details.

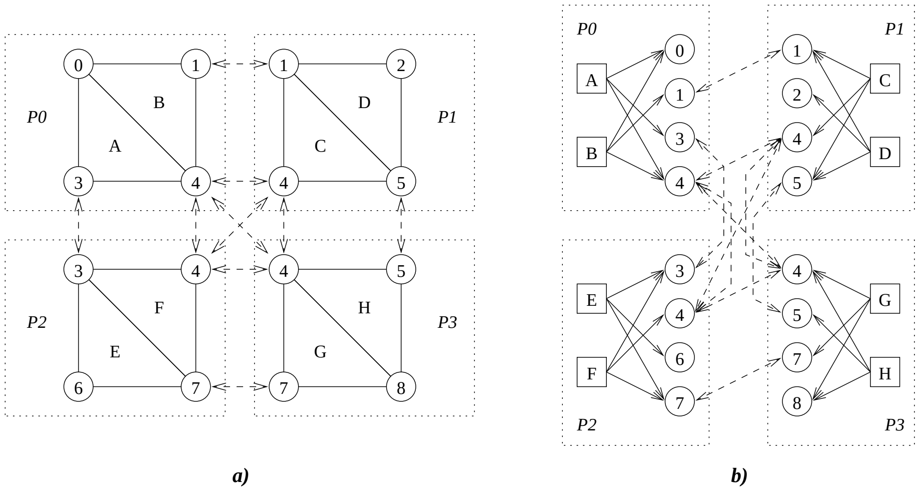


Figure 5: A square mesh partitioned by elements between four processors (a) and its internal representation using remote references (b).

If V_p is a node internal to the processor, then $Ref(V_p) = \varnothing$. A node in an internal boundary can be shared by more than two processors. Hence if V_p is a shared node then $1 \leq |Ref(V_p)| \leq P - 1$ where P is the number of processors. In a conforming mesh we expect that $|Ref(V_p)| \ll P - 1$ and usually $|Ref(V_p)| = 1$ for a shared node since most of the shared nodes are shared by only two processors in a 2-D mesh. The example in Figure 5 shows a mesh with 8 elements and 9 nodes. The node V_4 is shared by four processors P_0, P_1, P_2 and P_3 so $Ref(V_4^0) = \{(P_1, V_4^1), (P_2, V_4^2), (P_3, V_4^3)\}$ while $Ref(V_4^1) = \{(P_0, V_4^0), (P_2, V_4^2), (P_3, V_4^3)\}$. Figure 6 states for initializing the references.

There is no need to have more than one copy per node in each processor. Suppose that a processor P_i has two copies of the same node V_p^i and $V_p^{i'}$ so that $(P_i, V_p^{i'}) \in Ref(V_p^i)$. We can detect this condition because the reference points to a node in the same processor P_i. We then remove the copy $V_p^{i'}$ after updating all the references in other processors that point to $V_p^{i'}$ to point to V_p^i. For a similar reason we do not need or allow duplicate references in $Ref(V_p^i)$.

When a node V_p is created in an internal boundary between two processors P_i and P_j we initialize $Ref(V_p^i) = \{(P_j, V_p^j)\}$ and $Ref(V_p^j) = \{(P_i, V_p^i)\}$. Although at the end of refinement phase $|Ref(V_p^i)| = 1$ for each new node created in that phase, this might not hold after the load-balancing phase. It is possible that a new partition converts an internal node into a shared node and vice versa or that it modifies $Ref(V_p^i)$ so that it is shared by more than two processors.

The design of these references is highly influenced by the element partitioning

INPUT: $M(E, V)$ where M is a finite element mesh with a set E of elements and a set V of vertices.
– compute the dual $M^{-1}(E, W)$ of M where $W = \{(E_a, E_b) : E_a, E_b \in E, a \neq b,\ Adj(E_a) \cap Adj(E_b) \neq \varnothing\}$. W is a set of adjacent elements and $Adj(E_a)$ is the nodes of element E_a.
– partition M^{-1} into P regions using a graph partitioning algorithm such that $E_a \in \Pi_i$ if E_a is assigned to P_i where $\bigcup \Pi_i = E$ and $\Pi_i \cap \Pi_j = \varnothing \ \forall\ i \neq j$.
– $Nodes(\Pi_i)$ is the set of nodes corresponding to elements in Π_i. Note that is not required that $Nodes(\Pi_i) \cap Nodes(\Pi_j) \neq \varnothing$.
FOR each $V_p^i \in Nodes(\Pi_i)$ in parallel **DO**
  $Ref(V_p^i) = \varnothing$
END FOR
FOR each $V_p^i \in Nodes(\Pi_i)$ in parallel **DO**
  IF $E_a \in ElemAdj(V_p)$ and $E_a \in \Pi_j$ and $j \neq i$ **THEN**
    $Ref(V_p^i) = Ref(V_p^i) \cup (P_j, V_p^j)$
  END IF
END FOR

Figure 6: Computing the initial references in a parallel mesh.

method. Their main use is to maintain the connections between the different regions of the mesh as the mesh is partitioned between the processors. As will be shown in later sections, they provide a very flexible mechanism for maintaining a dynamic mesh. When a node is moved to a new processor it can use its reference list to find its copies in other processors. It can then send a message to these copies telling them to update their references to the new location. The references also simplify the task of assembling matrices and vectors from partially assembled ones as new nodes are created and moved at runtime because no assumption is initially made about origin and destination of these messages.

5.3 Using a parallel mesh for the solution of dynamic physical problems

In this paper we assume that we are given an initial coarse mesh M_0 at time $t = 0$ from which we find an initial partition Π^0. This partition is computed in a preprocessing step. We distribute the nodes and elements between the processors according to that partition and we compute the initial references using the algorithm in Figure 6.

Our algorithm for finding the solution of dynamic problems consists of four consecutive phases that we execute repeatedly. Figure 7 gives a high level outline of the program.

In the first phase we use numerical approximation techniques to find the solution of the partial differential equations by solving a system of linear equations. We solve this system using iterative methods. As we have mentioned

– find an initial partition Π^0.
– load the mesh using the partition Π^0.
– initialize the references $Ref(V_p^i)$ using the algorithm in Figure 6.
FOR $t < T$ **DO**
  – compute a solution.
  – refine/coarsen the mesh. For each new shared node V_p^i determine $Ref(V_p^i)$.
  – find a new partition Π^t.
  – migrate the elements and nodes according to Π^t. If a node V_p^i is moved from P_i to P_j then if V_p^k is another copy of V_p in P_k update $Ref(V_p^k) = ((Ref(V_p^k) - (P_i, V_p^i)) \cup (P_j, V_p^j)$ and set $Ref(V_p^j) = Ref(V_p^i)$.
END FOR

Figure 7: Outline of a general algorithm for computing the solution of dynamic physical system using a parallel mesh.

earlier we generally perform repeated matrix-vector products $Ab = c$ when we need to assemble matrices and vectors. All the effort in the following phases has the goal of improving the performance and quality of this phase.

At some time $t = t_k$ we decide that it is convenient to adapt the mesh so we start a refinement/coarsening phase. Using error estimates we select elements for refinement that we insert into R and if we select elements for coarsening we insert them in C. If the refinement of the elements in R creates a new shared node V_p in an internal boundary between two processors P_i and P_j we create the two local copies V_p^i and V_p^j and we initialize $Ref(V_p^i) = \{(P_j, V_p^j)\}$ and $Ref(V_p^j) = \{(P_i, V_p^i)\}$.

Since adaptation produces imbalances in the distribution of the work, we compute a new partition Π^t. If $\Pi^t \neq \Pi^{t-1}$ we need to migrate some elements and nodes to adequate the mesh to the new partition. This phase does not create new nodes or elements but it modifies the reference lists as nodes are moved to new processors.

6 Parallel mesh adaptation

Using the data structures presented in the previous section we now introduce an algorithm for adapting the mesh in a parallel computer. Let R be a set of elements selected for refinement and let R_i be the subset of the elements of R assigned to processor P_i. In this case $R = \bigcup R_i$ and also $R_i \cap R_j = \varnothing$ for $i \neq j$ because by using the element-partitioning method of assigning elements to processors each element is assigned to only one processor. Each processor has all the information it needs to refine in parallel its own subset R_i using a serial algorithm, but nonconforming elements might be created on the boundary between processor as suggested in Figure 8. In that example four elements are selected for refinement so $R_0 = \{E_a, E_e, E_f, E_g\}$ and $R_1 = \varnothing$. The refinement

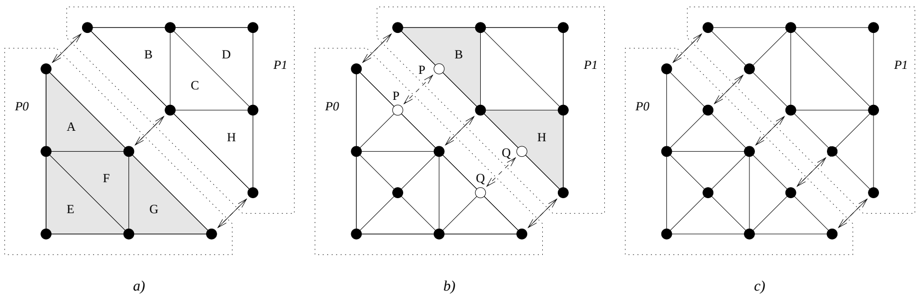


Figure 8: Propagation of refinement to adjacent processors. In (a) the elements E_a, E_e, E_f and E_g are selected for refinement. The refinement of these elements creates two nodes, V_p and V_q, in the boundary between P_0 and P_1. P_1 creates its local copies V_p^1 and V_q^1 and selects the nonconforming elements E_b and E_h for refinement (b). (c) shows the resulting mesh.

of E_a creates a node V_p^0 in an internal boundary between P_0 and P_1 and the refinement of E_g creates another shared node V_q^0. P_1 needs to create its local copies V_p^1 and V_q^1. It then marks the nonconforming elements E_b and E_h for refinement by inserting them in R_1 and invokes the serial refinement algorithm again.

6.1 Refinement collision

The parallel algorithm can run into two synchronization problems [3]. First, if processor P_i refines an element E_a and processor P_j refines an adjacent element E_b, it is possible that each processor could create a different node at the same position. In this case it is important that both processors do not consider them as two distinct nodes when assembling the matrices and vectors to compute the solution of the system and that the node incorporates the contributions of all the elements around it. Related to this problem is what processor P_i believes is a nonconforming element E_b in processor P_j might have already been refined there. Processor P_j needs to evaluate and update the propagation requests it receives before executing them. In this case P_j should insert a descendant of E_b in R_j.

These two problems are illustrated in Figure 9. In the top row we show a case where the receiving processor has already refined the element but further refinement is required. Initially E_a and E_b are selected for refinement. The refinement of E_a creates the shared node V_p^0. P_1 creates its copy of V_p but it then has to determine which of the children of E_b (E_{b_1} or E_{b_2}) should be inserted into R_1 for further refinement. In the bottom row the receiving processor should only update the reference rather than creating a new copy. Both P_0 and P_1 create shared nodes (V_p and V_q) in the same mesh location as the result of the

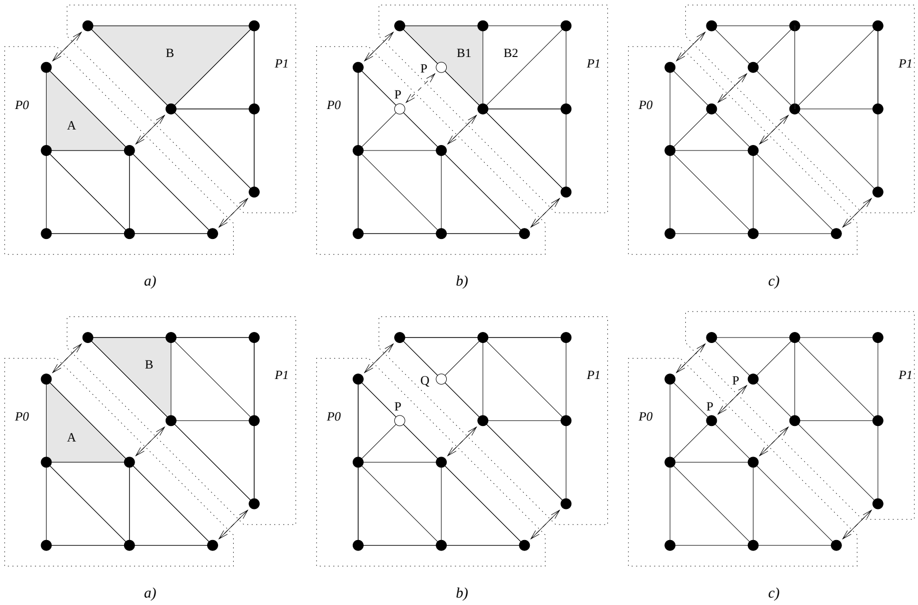


Figure 9: Refinement of adjacent elements located in different processors. In the top row two elements are selected for refinement (a). The refinement of E_a creates the shared node V_p (b). We then select E_{b_1} for further refinement (rather than E_b) (c). The bottom row shows another example (a) where two processors create shared nodes in the same position (b). In (c) we detect this problem and update the references.

refinement of E_a and E_b. We need to detect that both nodes are the same and update the references accordingly.

The solution to the synchronization problem is greatly simplified by using the nested elements of our multilevel algorithm. When an element E_b in processor P_j is refined into two or more elements E_{b_1} and E_{b_2} the element E_b is not destroyed as it would be the case in other refinement algorithms. Any message arriving to processor P_j from processor P_i with the instruction of making a copy of a shared node V_p in processor P_j (named V_p^j) that causes the refinement of the element E_b can be compared against the status of the element E_b. If the element E_b was already refined in the local phase (but processor P_i did not know about this), then the element E_b might not need to be refined again. If the node V_p was already created in the local phase of processor P_j then a reference is added pointing to its copy V_p^i in processor P_i. If the refinement of the element E_b did not cause or was not caused by the creation of the shared node V_p (for example the refinement was done dividing another edge as in the top row of Figure 9), then its children E_{b_1} and E_{b_2} are evaluated and the one that shares the internal boundary between P_i and P_j is marked for refinement

R_i is the set of elements selected for refinement in P_i.
WHILE $R_i \neq \varnothing$ **DO**
– extract an element E_a from R_i.
– refine E_a using a serial refinement algorithm.
FOR each shared node V_p^i created in an internal boundary between P_i and P_j **DO**
Send a message from P_i to P_j requesting the creation of a shared node V_p^j in P_j. If a node V_p^j already exists, then return a reference to it. Otherwise, create the node V_p^j, determine the element to refine, and insert it into R_j. Finally return its reference to P_i.
END FOR
END WHILE

Figure 10: Avoiding refinement collisions in a parallel mesh.

using the shared node V_p^j.

The pseudocode for this algorithm is shown in Figure 10 but there are a some details that they are not explained there. First, we do not send a message for each individual node because of the high cost of sending messages. Instead we first refine all the elements in R_i keeping track of the shared nodes that P_i creates as a result of refining elements in R_i. Once $R_i = \varnothing$, P_i sends the messages to its adjacent processors and listens for propagation messages from them. If it receives such a message it creates the new shared nodes and inserts the nonconforming elements into R_i.

To determine which element to refine we use $ElemAdj(V_p)$. Suppose that the refinement of an element E_a in P_i creates a shared node V_p in a boundary between P_i and P_j. This new node is created at a midpoint between two other shared nodes V_q and V_r. Note that $(P_j, V_q^j) \in Ref(V_q^i)$ and also $(P_j, V_r^j) \in Ref(V_r^i)$. We use these references to send a message from P_i to P_j. When P_j receives this message, it determines the unrefined element $E_b \in ElemAdj(V_q^j) \cap ElemAdj(V_r^j)$ and inserts it into R_j.

As it can be easily seen, the parallel algorithm is not a perfect parallelization of the serial one and it can result in a different mesh. The serial Rivara's algorithm [1] and [2] first selects an element E_a from R. It then continues refining all the nonconforming elements that result from the refinement of E_a before proceeding with another element from R. In our parallel implementation we ignore this serialization. We approximate the serial algorithm within each processor as much as possible but do not impose it across processor boundaries. We claim that this modification does not affect the quality of the refinement.

6.2 Termination detection

The algorithm for detecting the termination of parallel refinement is based on a general termination algorithm in [4]. A global termination condition is reached when no element is marked for refinement, so if R is the set of all the elements selected for refinement, then the algorithm finishes when $R = \varnothing$. This global termination condition implies a local termination condition for processor P_i that holds when $R_i = \varnothing$. We assume that the refinement is started in one special processor referred to as the *coordinator*, P_C. To simplify the explanation we assume initially that the refinement does not propagate cyclically from processor P_i to processor P_j and then from processor P_j back to processor P_i. We will show later that this is not a reasonable restriction but it does not affect the algorithm significantly.

The algorithm for detecting termination uses two basic kind of messages:

- a *refine* message *Refine-Msg*(i, j) sent from a source processor P_i to processor P_j is used to request the refinement of one or more elements of processor P_j. We will specify the contents of this message later but let's assume for now that it can either indicate the elements selected for refinement (if the message is sent by the coordinator) or it can include a reference to a shared node. If P_j receives a *Refine-Msg*$(i, j) = \{V_p^i\}$ it creates the node V_p^j, it initializes $Ref(V_p^j) = \{(P_i, V_p^i)\}$ and inserts the unrefined element $E_a \in Adj(V_p)$ into R_j. Note that at this stage V_p^i has no reference to V_p^j. To update $Ref(V_p^i)$ we use the next type of message.

- an *update* message *AddRef-Msg*(j, i) is returned from P_j to P_i for each refine message sent from P_i to P_j to indicate the completion of the requested refinement. This message also includes the necessary information to update the references to the nodes shared between P_i and P_j. When P_i receives an *AddRef-Msg*$(j, i) = \{V_p^j\}$ it inserts (P_j, V_p^j) into $Ref(V_p^i)$. If P_i is the coordinator we return *AddRef-Msg*$(j, C) = \varnothing$.

The coordinator sends at $t = 0$ a *Refine-Msg*(C, i) message to one or more processors P_i indicating that the refinement phase has started. The initiator can explicitly select the elements for refinement or it can instruct the processors to select the elements based on an adaptation criteria. Processor P_i then executes the serial refinement algorithm on these marked elements, possibly sending *Refine-Msg*(i, j) messages to neighboring processors P_j when a node V_p^i is created in an internal boundary between processors P_i and P_j.

The local termination condition holds for processor P_i when no more elements are marked for refinement. When this condition holds, processor P_i does not generate new *Refine-Msg* messages and it is not waiting for any *AddRef-Msg* messages. Also processor P_i does not insert new elements into R_i until a *Refine-Msg*(j, i) message arrives from some other processor P_j. In this case, new nodes are created in the internal boundary as instructed in the message and the corresponding elements are selected for refinement by inserting them into R_i. Then processor P_i executes the serial refinement algorithm, which might cause further propagation to other processors. An example is shown in Figure 11 where P_C sends an initial *Refine-Msg* to the other processors. P_0 and P_1 complete their work without propagation so they return a *AddRef-Msg* message to

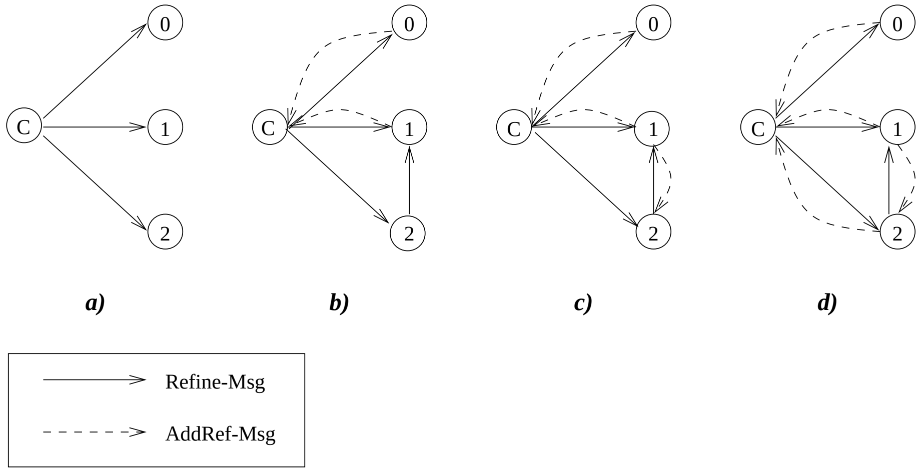


Figure 11: Parallel refine algorithm. In (a) the initiator sends a *Refine-Msg* to each other processors. Processors P_0 and P_1 return immediately a *AddRef-Msg* to the initiator but the refinement in processor P_2 propagates to P_1 so P_2 sends a *Refine-Msg* to P_1 (b). After P_1 returns a *AddRef-Msg* to P_2 (c), P_2 returns its *AddRef-Msg* to the initiator (d).

P_C. On the other hand the refinement of elements in P_2 propagates to P_1 so P_2 sends a *Refine-Msg*$(2,1)$ message to P_1. P_1 completes this request without further propagation so it returns to P_2 which in turn returns an *AddRef-Msg*$(2, C)$ message to P_C.

We say that the parallel refinement terminates (the global termination condition holds) at some t if:

- $R_i = \varnothing$ for each processor P_i at time t.
- there is no *Refine-Msg* or *AddRef-Msg* in transit at time t.

The termination detection procedure is based on message acknowledgments. In particular *Refine-Msg*(i, j) messages received by processor P_j from processor P_i are acknowledged by P_j by sending to P_i a *AddRef-Msg*(j, i) message. These messages return the references to the newly created vertices so that if V_p^i is a vertex in processor P_i over a shared edge that caused a propagation to processor P_j and V_p^j is its copy in processor P_j, a reference to V_p^j is added at V_p^i and vice versa.

A processor P_i can be in two possible states: the *inactive* state and the *active* state. While in an inactive state P_i can send no *Refine-Msg* or *AddRef-Msg* but it can receive messages. If it receives a receives a *Refine-Msg*(j, i) from another processor P_j while in an inactive state it moves from the inactive to the active state. It creates the shared nodes as stated in the message and proceeds to refine the nonconforming elements. The message *Refine-Msg*(j,i) is called the *critical* message because it caused the refinement that P_i is performing and P_j is the parent of processor P_i.

In the active state, while a processor P_i is refining some of its elements, the refinement can propagate to a neighbor processor P_k requiring another *Refine-Msg*(i,k) message to it. An active processor becomes inactive at the first time t for which the following conditions hold:

- no new *Refine-Msg* message is received by the processor at time t.
- there are no elements marked for refinement in processor P_i (the local termination condition holds).
- the processor has transmitted prior to t an *AddRef-Msg* message for each *Refine-Msg* message it has received except for the critical message.
- the processor has received prior to t a *AddRef-Msg* message for each *Refine-Msg* message it has transmitted.

Using this definition, the local termination condition might hold in processor P_i ($R_i = \varnothing$) but processor P_i might be in an active state waiting for a *AddRef-Msg*(j,i) from processor P_j if the refinement of the elements of P_i caused the refinement to propagate to processor P_j and P_j has not yet responded. When a processor becomes inactive it returns a *AddRef-Msg* message to the processor that originally sent its critical message.

Initially, at time $t = 0$, the coordinator is active and all other processors are inactive. Furthermore, at $t = 0$, the local termination condition holds at all processors except the coordinator. It can be seen that if a processor is inactive at time t, the following rules hold:

- its local termination condition holds at t.
- it has transmitted an *AddRef-Msg* for all the *Refine-Msg* messages it has received prior to t.
- it has received *AddRef-Msg* messages for all *Refine-Msg* messages it has transmitted prior to t.

If the processor is active at time t, at least one of the above conditions is violated. We say that global termination is detected when the coordinator becomes inactive. In the case of the coordinator only the last of the previous rules is relevant as it has no local elements to refine. The coordinator will receive an *AddRef-Msg* message from all the processors P_i only when all the processors are inactive. In this case there are no elements marked for refinement and there are no other messages in the network.

This algorithm works if the propagation does not generate cycles. As shown in Figure 12 it is possible to design a mesh where the refinement propagates back to processor P_0. In that example P_0 refines an element E_a creating a

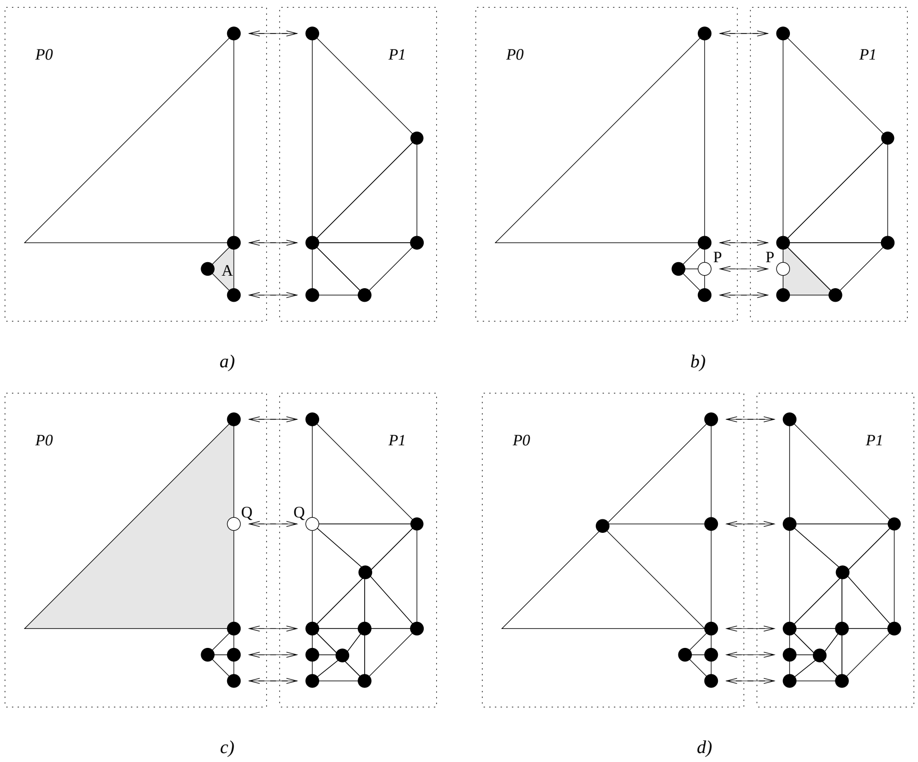


Figure 12: A propagation cycle. P_0 has initially one element marked for refinement (a). The refinement propagates to P_1 (b) and then comes back to P_0 (c). (d) shows the final mesh.

shared node V_p^0. It then sends a *Refine-Msg*$(0,1)$ to P_1. P_1 creates its copy of the shared node and proceeds to refine the nonconforming elements but before P_1 is ready to return a *AddRef-Msg*$(1,0)$ a new shared node V_q^1 is created in the boundary between P_1 and P_0. In this case P_1 sends a new *Refine-Msg*$(1,0)$. When P_0 performs all the required refinements it returns a *AddRef-Msg*$(0,1)$ to P_1 which in turn returns a *AddRef-Msg*$(1,0)$ to P_0 corresponding to the initial message. In this example is not easy to detect which one is the parent or the child processor. It also shows that the refinement of some meshes can have a cycle. It is possible to extend the idea to a long but finite sequence of *Refine-Msg* messages through two processors before being ready to return a *AddRef-Msg*. Fortunately we can modify the previous algorithm to deal with these problems.

In the active state a processor P_i can receive not only *AddRef-Msg* messages from its neighbors but also new *Refine-Msg* messages from other processors P_j. These new *Refine-Msg* might cause further propagation. Now there is not just one critical message for processor P_i but there is still only one critical message for

```
FOR each processor P_i DO
  send Refine-Msg(C, i) indicating elements to refine.
END FOR
FOR each processor P_i DO
  wait for an AddRef-Msg(i, C).
END FOR
FOR each processor P_i DO
  send Continue-Msg(C, i) to finish the refinement phase.
END FOR
```

Figure 13: Detecting the termination of the refinement phase (Coordinator algorithm).

each of the *Refine-Msg* messages that processor P_i transmits to other processors. We modify the *Refine-Msg* message to include a unique sequence number for each processor. We also modify the *AddRef-Msg* message to return the number of the *Refine-Msg* that caused the refinement.

All the critical messages are added to a table of critical messages. When processor P_j sends back a *AddRef-Msg* message it needs to identify which critical message in processor P_i caused the propagation to P_j using the sequence number. When a critical message in a processor P_i receives a *AddRef-Msg* message for all the propagations it caused, then processor P_i removes the critical message from the table and it sends back a *AddRef-Msg* message to the processor that initially sent that critical message to it. The processor P_i is in the inactive state if $R_i = \varnothing$ and it has no critical messages in its table. The pseudocode for the algorithm executed by the coordinator is shown Figure 13 while Figure 14 has an outline of the program executed by all the remaining processors. This pseudocode is explained below.

Initially P_C sends a *Refine-Msg*(C, i) to all the other processors P_i to start the refinement phase. These messages are used to select the elements in R_i to be refined in P_i. Each P_i returns a *AddRef-Msg*(i, C) once they have refined these elements and has also received a *AddRef-Msg*(k, i) for each *Refine-Msg*(i, k) produced by the propagation of the refinement to P_i. The algorithm uses a new type of message:

- a *continue* message *Continue-Msg*(i, j) sent from the initiator to every other processor is used to inform them that the refinement phase concluded and that they can continue to the next phase.

Once P_C has received a *AddRef-Msg* from each other processor it broadcasts a *Continue-Msg*.

Each processor P_i starts the refinement phase waiting for a message. If it receives a *Continue-Msg* from the initiator it knows that it can proceed to the next phase. If the message is a *Refine-Msg*(k, i) it inserts the elements indicated in the message in R_i and refines it using the serial algorithm. Rather than having

```
seq = 0
WHILE true DO
  wait for a message msg.
  IF msg = Continue-Msg(j, i) THEN
    return.
  ELSE IF msg = Refine-Msg(j, i) THEN
    set seq++, table[seq].critical = msg and table[seq].count = 0
    FOR each element E_a ∈ msg DO
      create the shared nodes and insert E_a in R_i.
    END FOR
    refine the elements in R_i
    FOR each shared node V_p^i created in an internal boundary between P_i and
    P_k DO
      send Refine-Msg(i, k) containing seq and the elements to refine.
      table[seq].count++
    END FOR
    IF table[seq].count = 0 THEN
      return AddRef-Msg(i, j) as msg did not cause refinement to other pro-
      cessors.
    END IF
  ELSE IF msg = AddRef-Msg(j, i) THEN
    seq is the sequence number returned by msg. set table[seq].count - -
    IF table[seq].count = 0 THEN
      send AddRef-Msg(i, j) to P_j where P_j sent the message stored in ta-
      ble[seq].critical.
    END IF
  END IF
END WHILE
```

Figure 14: Detecting the termination of the refinement phase.

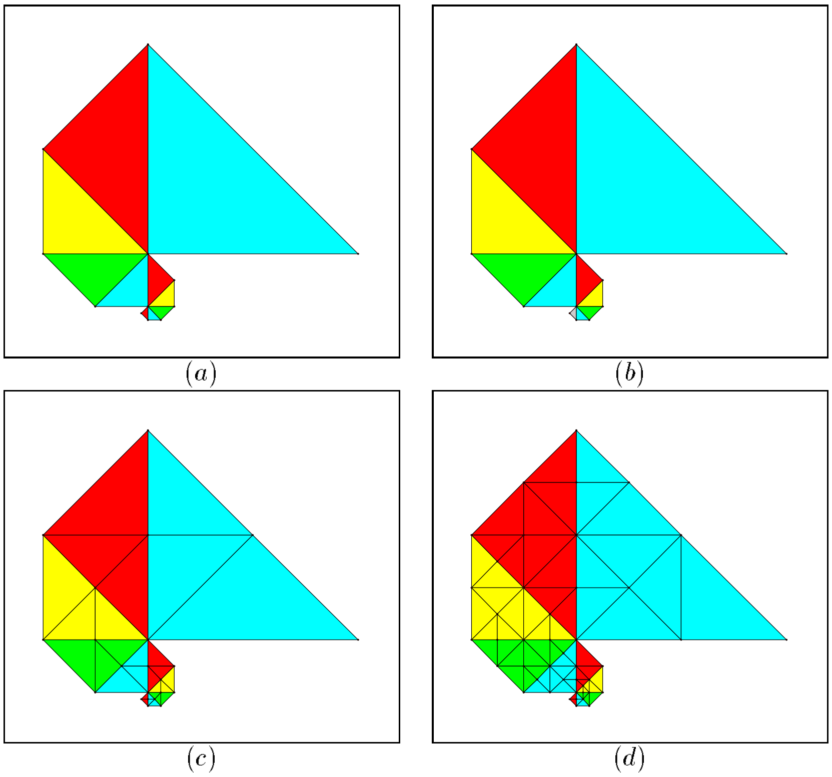


Figure 15: Propagation of the refinement. In (a) we show an arbitrary mesh distributed in 4 processors. The refinement of an element (b) causes the refinement to come back to the processor (c). If we repeat this operation we obtain the mesh in (d).

one critical message now P_i can have several critical messages sent by the same or different processors. P_i gives them a sequence number and stores them in a table. If the refinement of the elements in R_i creates shared nodes in a boundary between P_i and P_j then P_i sends a *Refine-Msg*(i, j) message to P_j. P_i keeps track of how many of these *Refine-Msg*(i, j) it sends for each critical *Refine-Msg* it receives. Once it has received a *AddRef-Msg*(j, i) for each *Refine-Msg*(i, j) it can send back a *AddRef-Msg*(i, k) response to P_k. P_i continues to listen for messages until it receives a *Continue-Msg* from the coordinator. Figure 15 shows an example where the refinement propagates cyclically between processors.

7 Load balancing

In this section we present a strategy for repartitioning and rebalancing the mesh. We first explain serial multilevel refinement algorithms. We then introduce a new highly parallel repartitioning method called the Parallel Nested Repartitioning (PNR) algorithm which is fast and gives high quality partitions.

In Section 7.2 we explain a mesh migration algorithm. This algorithm receives as input the partition obtained from the repartitioning of the mesh and migrates the elements and nodes according to this partition.

7.1 The mesh repartitioning problem

While the PNR repartitioning algorithm is based on the serial multilevel algorithms presented in [15], [20] and [18] it also makes use of the refinement history to achieve great reductions in execution time and an improvement in the quality of the partitions produced. General multilevel algorithms partition the mesh by constructing a tree of vertices. They create a sequence of smaller graphs by collapsing vertices, then partition a suitable small graph and finally reverse the collapsing process to produce a partition of the larger graphs.

The Parallel Nested Repartitioning algorithm can be divided into a serial phase and a parallel phase. When the graph is small enough to fit into one processor we use a serial refinement algorithm to partition the graph. When the mesh is very large as in the case of a highly refined mesh we collapse vertices in parallel. The PNR method differs from other methods in that it uses the refinement history of the mesh to collapse the vertices while other methods use maximum matchings or independent sets. As a consequence we are able to collapse vertices locally in the parallel phase without any communication overhead unlike other methods. Our tests show that by using the refinement history we obtain partitions that are almost always of higher quality than those obtained by the multilevel algorithms yet PNR is very fast. For simplicity we assume that the initial mesh fits into one processor and marks the transition between the serial and the parallel phase. In Section 7.1.5 we discuss possible generalizations of this method.

7.1.1 The serial Multilevel Graph Partitioning algorithms

The code for a standard serial Multilevel Partitioning Algorithm is sketched in Figure 16. In general serial multilevel algorithms perform the partitioning of a mesh in three phases:

- in the *coarsening* phase these algorithms construct a sequence of graphs $G_0, G_1, \ldots G_k$ such that the $|G_i| < |G_{i-1}|$ by collapsing adjacent nodes or contracting edges. This contraction is implementing by finding a maximal independent set [16] or a maximal matching [20]. Given a graph $G(U, F)$ where U is a set of vertices and F is a set of edges then $U' \subseteq U$ is an independent set of G if for all $v \in U', (v, w) \in F \Rightarrow w \notin U'$. An independent set U' is maximal if the the addition of any vertex in U' would make it no longer an independent set. A matching of G is a set $F' \subseteq F$ od edges such that no two of which are incident on the same verex. A mathing F' is maximal if the addition of an edge in F' would make it

compute the weighted graph $M_0^{-1}(E, W) = G_0$.
WHILE $|G_i| > K$ **DO**
compute a coarser graph G_{i+1} by collapsing the vertices of G_i.
END WHILE
partition the coarsest graph G_k.
FOR each level i in the refine tree from k downto 0 **DO**
project the partition of G_i to G_{i-1}.
improve the partition of G_{i-1} using local heuristics.
END FOR

Figure 16: Serial Multilevel Partitioning algorithm.

no longer a matching. A contraction operation is repeated until $|G_i|$ is smaller than a defined constant K.

- in the *partitioning* phase standard multilevel methods find a partition Π of the graph G_k using any one of a number of different graph partitioning algorithms such as Recursive Spectral Bisection [15]. Note that typically $|G_k| \ll |G_0|$ so their use of RSB is generally not very expensive.
- in the *uncoarsening* phase these methods project the partition found for G_i to the graph G_{i-1} by reversing the collapsing process. Assume that two or more vertices v and w in the graph G_{i-1} are collapsed to form a vertex u in G_i in the *coarsening* phase. If u is assigned to processor P_q then both r and s are initially assigned to P_q. After projecting the partition to G_{i-1}, they typically perform local heuristics such as Kernighan and Lin [32] on each G_{i-1} for the purpose of improving the quality of the partition.

To implement this algorithm on a parallel computer note that for each level of the coarsening phase we need to compute either an independent set or a matching of the graph. This implies that for each G_{i-1} we will need to send messages to insure that two adjacent processors do not select adjacent vertices of the graph at the same time, an operation that can be very time consuming. As we will show, our algorithm uses a different heuristic for collapsing the vertices of the graph that does not require synchronization at each level.

7.1.2 General repartitioning of an adapted mesh

In Section 5.1 we explained that our meshes are partitioned by elements. That is, given a mesh $M(E, V)$ where E is a set of elements and V is a set of vertices we construct a partition $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_P\}$ such that each element $E_a \in E$ is assigned to only one partition Π_i. This is done by creating a graph $M^{-1}(E, W)$ that is the dual of M where W is a set of pairs of adjacent elements: $(E_a, E_b) \in W$ if and only if E_a and E_b are adjacent. Thus the elements in E are the vertices of the graph M^{-1} and the pairs in W are its edges. A partition of the vertices of M^{-1} is a partition of the elements of the mesh M.

In order to contract the graph while preserving its global structure we associate weights to each element E_a and each pair of adjacent elements (E_a, E_b). Given the graph M^{-1} we define $ElemWeight(E_a)$ to be the number of descendants of E_a in the fine mesh (or 1 if E_a has no children). We also define $EdgeWeight(E_a)$ to be the number of edges between the descendants of E_a in the fine mesh. That is $EdgeWeight(E_a) = \sum_{E_b, E_c \in M_t}(E_b, E_c)$ such that E_b and E_c are the lowest level descendants of E_a and they are adjacent to each other. For each pair $(E_a, E_b) \in W$ we define $Weight(E_a, E_b)$ to be the number of edges between the descendants of E_a and E_b in the fine mesh. Note that if both E_a and E_b are unrefined then $Weight(E_a, E_b) = 1$.

Although we defined *ElemWeight*, *EdgeWeight* and *Weight* based on our multilevel elements of Section 3 we could also define them recursively for any mesh. Given $M^{-1}(E, W)$ we define $ElemWeight(E_a) = 1$, $EdgeWeight(E_a) = 0$ and $Weight(E_a, E_b) = 1$ if E_a and E_b are adjacent and 0 otherwise. If we collapse two or more elements E_a and E_b into one element E_c then:

$$ElemWeight(E_c) = ElemWeight(E_a) + ElemWeight(E_b)$$

$$EdgeWeight(E_c) = EdgeWeight(E_a) + EdgeWeight(E_b) + Weight(E_a, E_b)$$

In a similar way we can define $Weight(E_c, E_d)$ for two collapsed elements E_c and E_d.

The goal of the coarsening phase in the partitioning algorithm is to approximate cliques. A refined vertex E_a of the graph M^{-1} approximates a clique if

$$EdgeDensity(E_a) = \frac{(2 \times EdgeWeight(E_a))}{(EdgeWeight(E_a)(EdgeWeight(E_a) - 1))}$$

is close to 1 and it does not approximate a clique if it is close to 0. Intuitively, if a vertex E_a has a a large edge density it will not be cut in half by the bisection in the partition of the contracted graph.

7.1.3 The Parallel Nested Repartitioning (PNR) algorithm

The partitioning algorithm that we discuss in this section incorporates the idea that the fine mesh M_t at time t was obtained as a sequence of refinements of a coarse initial mesh M_0. The mesh M_t includes all the elements E_a such that $Children(M_a) = \varnothing$ at time t. We define $|M|$ as the number of elements in the mesh. We assume that $|M_0| < |M_t|$ but in general $|M_0| \ll |M_t|$. Note that it is possible to have an element $E_a \in M_0 \cap M_t$ if E_a is not refined.

Figure 17 shows an example of an initial mesh M_0 and the refined mesh M_t at time t. The amount of work for processor P_0 is far larger than the amount of work of the other processors. The goal of the repartitioning algorithm is to rebalance the work so each processor has approximately the same number of elements.

The PNR algorithm uses the information that M_t was obtained as a sequence of refinements. Rather than computing directly a partition of M_t it computes

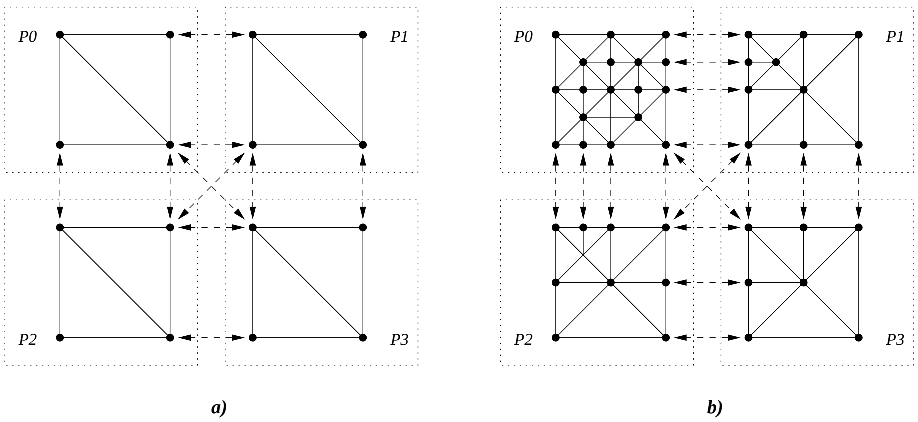


Figure 17: The Parallel Nested Repartitioning algorithm. (a) shows the initial mesh M_0 and (b) shows the mesh M_t at the beginning of the partitioning phase.

a partition of M_0 and then projects this partition to M_t. The notion is that in the coarsening phase we do not need to find a matching or independent set to collapse the children of refined elements. Instead we use the refinement tree to collapse the descendants of each element of the coarse mesh M_0. In Section 9.6 we compare the PNR with the serial multilevel algorithm. By using the refinement history we are able to obtain better partitions at a lower cost than the standard methods.

Our PNR algorithm allows for a very natural parallel implementation. Is is possible to compute the *ElemWeight*, *EdgeWeight* and *Weight* of M_0^{-1} in parallel using only local information. We then send the graph M_0^{-1} to a common processor P_P which partitions the reduced graph M_0^{-1}. At this point all the other processors wait until P_P sends back a message informing them of which elements to migrate. Finally, the migration is performed by moving fully refined subtrees. The partitioning algorithm will inform a processor P_i of the elements $E_a \in M_0$ to move to another processor P_j. It is assumed that P_i will not only send E_a but also all its descendants to P_j. The intention is not to break partition trees to simplify the unrefinement of the elements. In the rest of this section we will explain the algorithm in more detail using the example shown in Figure 17. The pseudocode for the PNR algorithm is shown in Figure 18 and explained below.

As we explained earlier, our Parallel Nested Repartitioning algorithm for mesh partitioning can be divided into a *parallel* phase and a *serial* phase:

- we construct in parallel the weighted graph M_0^{-1}. Communication is not required at this point. Each processor P_i computes the weights of each element $E_a \in M_0$. This is done using a simple recursive algorithm. In the same way it computes the weight of each edge $W = (E_a, E_b)$. Once

– in parallel compute *ElemWeight*(E_a), *EdgeWeight*(E_a) and *Weight*(E_a, E_b) for each element E_a and each pair of adjacent elements $(E_a, E_b) \in M_0^{-1}$.
– each processor P_i sends its portion of M_0^{-1} and the weights to a common processor P_P.
– P_P receives sections of M_0^{-1} from each processor and computes a partition Π of M_0^{-1}. This can be done using RSB or a serial multilevel algorithm.
– P_P returns Π to each processor P_i.
– P_i migrates the elements and nodes according to Π.

Figure 18: The parallel Nested Repartitioning algorithm.

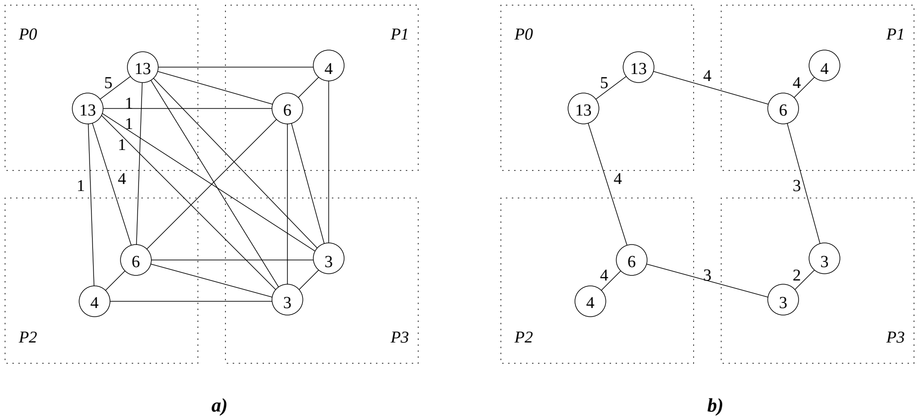


Figure 19: The Parallel Nested Repartitioning algorithm. (a) shows the graph G where there is an edge in G between each two elements in M_0 that share a node. (b) shows the graph G where there is an edge in G between each two elements in M_0 that share a common edge.

P_i obtains its portion of M_0^{-1} it sends it to P_P for the serial part of the algorithm. The M_0^{-1} graph for the mesh in Figure 17 is shown in Figure 19. We consider two ways of defining set W:

- $W = \{(E_a, E_b) : E_a, E_b \in M_0,\ E_a \text{ and } E_b \text{ have a common vertex}\}$.
- $W = \{(E_a, E_b) : E_a, E_b \in M_0,\ E_a \text{ and } E_b \text{ have a common edge}\}$.

- once P_P receives a message for each processor P_i it partitions the reduced graph M_0^{-1} using a serial partitioning algorithm. As $|M_0^{-1}|$ is assumed to be relatively small we can use at this stage algorithms that would be considered too expensive to apply to the refined mesh. The result of this partition is shown in Figure 20 (a).

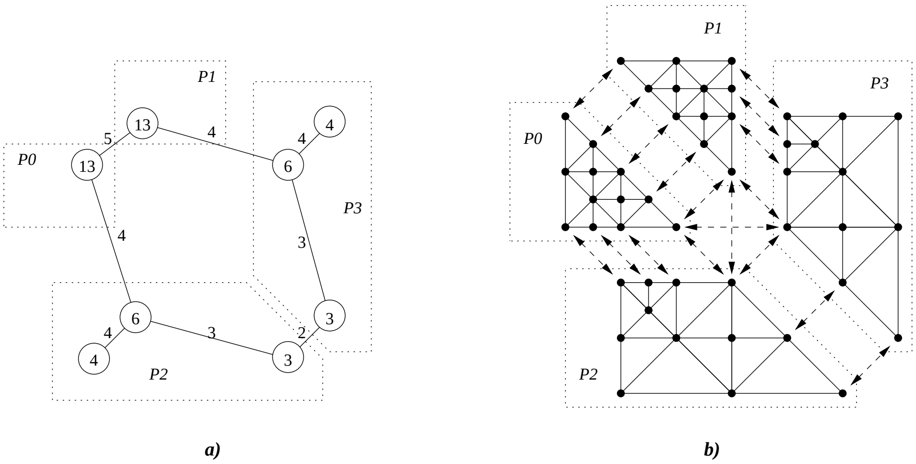


Figure 20: The Parallel Nested Repartitioning algorithm. (a) shows the partition Π of M using the PNR algorithm. Finally we use the migration algorithm to migrate elements and nodes to their destination processors. (b) shows the resulting mesh.

- finally we resume the parallel phase. P_P sends a message to each processor P_i informing it of which elements to migrate. P_i executes the migration algorithm described in the following section to distribute the mesh as shown in Figure 20 (b).

Our method does not require that the complete fine mesh be in one processor in order to compute the partition. It is sufficient that the coarse initial mesh is small enough to fit into one processor. The refined mesh can be of an arbitrary size.

7.1.4 Partitioning of an initial mesh

We have yet to explain how to compute the partition of M_0^{-1} in P_P. In theory we can use any serial graph partitioning algorithm without affecting the structure of the PNR algorithm but in practice we use one of two approaches.

In one case P_P spawns a new process that calls *Chaco* [21]. This process finds a partition of M_0^{-1} using the multilevel algorithm (or any other partitioning algorithm supported by *Chaco*) and returns it to P_P.

In the other case P_P computes the partition of M_0^{-1} directly. We initially find a matching of the graph, defined to be a set of edges such that no two edges in that set are incident in the same vertex. Once we find this matching we collapse pairs of vertices to form a new vertex. As a consequence, for $1 \leq i \leq k$ we create a subgraph $G_i(E_i, W_i)$ of $G_{i-1}(E_{i-1}, W_{i-1})$ where $|E_i| < |E_{i-1}|$. We also compute $ElemWeight(E_c)$, $EdgeWeight(E_c)$ and $Weight(E_c, E_d)$ for each element E_c and each pair (E_c, E_d) of adjacent elements in G_i.

We choose a matching that has an approximate maximal edge density. We approximate the matching by using a randomized algorithm. We select in random order an unmatched vertex r and we determine for each unmatched neighbour s of r the *EdgeDensity* of a vertex u formed by collapsing r and s. Then r is collapsed with the neighbour with which it has the largest edge density.

We then partition the graph G_k using a partitioning algorithm. In our tests we used Recursive Spectral Bisection. We usually call *Chaco* for this purpose. This is very fast since $|E_k|$ is small.

Finally we uncoarsen the graph for each level $k > i > 0$. At this time we also improve the partition using local heuristics that are a variation on Kernighan-Lin [32]. We compare pairs of elements assigned to different processors and if we find that there is an improvement in the quality of the partition, flip them. While these algorithms have been implemented, performance results are not reported because the method provides at least equally good results.

7.1.5 Improving the PNR algorithm

In this section we discuss three possible improvements to the PNR algorithm. Two of them are generalizations of the PNR algorithm while the third one is a discussion of parallel heuristics to improve the quality of the partition.

We assumed that the transition between the parallel phase and the serial phase is given by the initial mesh M_0. This does not always have to be the case. If an element $E_a \in M_0$ is highly refined, we can send the children of E_a rather than E_a to P_P or some of its descendants.

Although we send the full mesh M_0 to P_P with all the weights each time we repartition the mesh this is not necessary. If we assume that the serial partition of M_0 is computed using a serial multilevel algorithm then we can just compute the tree once and store it in P_P. To repartition the mesh P_i needs to send to P_P only the changes of the weights produced as the result of the refinement and coarsening of the mesh. In this way we are extending the PNR to graphs that are not obtained as the result of a refinement process.

In the migration algorithm explained in the next section we migrate fully refined trees. This means that at every time step t if an element $E_a \in M_0$ is assigned to processor P_i then all the descendants of E_a are also assigned to P_i. For this reason we have not yet implemented parallel heuristics such as the MOB heuristic [9] to try to improve the quality of the partition.

7.2 Using remote references for work migration

Although we demonstrated in the previous section how to compute a partition Π^t that balances the work, at this stage of the computation the mesh is still distributed according to an unbalanced partition Π^{t-1}. In this section we present an algorithm that migrates elements and nodes between processors. If $\Pi^t \neq \Pi^{t-1}$, then there is at least one element E_a such that $E_a \in \Pi_i^{t-1}$ and

$E_a \in \Pi_j^t$ where $i \neq j$. Remember that we assume that the mesh is partitioned by elements so that the Π_i partitions are disjoint, $E_a \notin \Pi_j^{t-1}$ and $E_a \notin \Pi_i^t$. To adjust the mesh to the Π^t partition we need to move E_a from P_i to P_j. Let's assume that the vertices of E_a are $Adj(E_a) = \{V_{p_1}, \ldots, V_{p_n}\}$.

Our algorithm considers several cases that depend on P_j having a local copy of these nodes or if they are included in the message to P_j:

- for each node $V_p \in Adj(E_a)$, if $(P_j, V_p^j) \notin Ref(V_p^i)$ (so V_p is not a shared node between P_i and P_j at time $t-1$) then we need to create the node V_p^j in P_j and then use this node to create the element E_a in P_j.

- otherwise $(P_j, V_p^j) \in Ref(V_p^i)$ (so V_p is a shared node between P_i and P_j at time $t-1$ and P_j has a local copy V_p^j) then we should not create the V_p^j node again. When P_i sends the element E_a to P_j, it also includes the reference (P_j, V_p^j) instead of the node V_p. Then P_j can use V_p^j to create E_a. This condition has an important implication: processor P_j cannot delete its copy of V_p^j until it has received all the elements, even if processor P_j has already sent the only element E_c that points to V_p^j to another processor P_k because some other processor P_i might expect P_j to have a copy of V_p.

- if processor P_i sends more than one element E_a and E_b to P_j and there is a node $V_p \in Adj(E_a) \cap Adj(E_b)$ (so V_p is a vertex of both E_a and E_b) then only one copy V_p^j should be created in P_j and both elements E_a and E_b should refer to it in the destination processor.

- now if two processors P_i and P_k send the elements E_a and E_b respectively to processor P_j where elements E_a and E_b are adjacent elements and there is a shared node $V_p \in Adj(E_a) \cap Adj(E_a)$ so $(P_k, V_p^k) \in Ref(V_p^i)$ and $(P_i, V_p^i) \in Ref(V_p^k)$ then P_j should detect that V_p^i and V_p^k are two copies of the same node. In this case P_j should create only one copy V_p^j for both elements E_a and E_b.

- finally if processor P_i sends an element E_a to another processor P_j and P_k sends an element E_b to P_l and E_a and E_b are adjacent elements in two different processors that share a common node V_p then we should insure that $(P_l, V_p^l) \in Ref(V_p^k)$ and $(P_k, V_p^k) \in Ref(V_p^l)$ (so V_p^k and V_p^l refer to each other).

In the migration phase we use three different kinds of messages:

- a *move* message $\textit{Move-Msg}(i,j) = \{E_a\}$ is used to migrate an element E_a from a source processor P_i to a destination processor P_j. We also assume that this message has two other fields. The first field $\textit{Move-Msg}(i,j).\textit{nodes}(E_a)$ contains the vertices of the element E_a. For each node $V_p \in Adj(E_a)$, if P_j has a local copy of V_p^j then only the reference (P_j, V_p^j) is included in the message. Otherwise we send the node V_p.

We also set $Ref(V_p^i) \cup (P_i, V_p^i)$ to initialize V_p^j. The other field, *Move-Msg*$(i,j).ref(V_b)$, contains the references to V_p in the other processors.

- an *add reference* message *AddRef-Msg*$(i,j) = \{(V_p^j, V_p^i)\}$ is used to add a reference to node V_p^i in processor P_j. In this case we set $Ref(V_p^j) = Ref(V_p^j) \cup (P_i, V_p^i)$. This is essentially the same kind of message used for refining the mesh.
- a *delete reference* message *DelRef-Msg*$(i,j) = \{(V_p^j, V_p^i)\}$ is used to remove a reference to the node V_p^i in P_j. So $Ref(V_p^j) = Ref(V_p^j) - (P_i, V_p^i)$.

In the rest of this section we discuss a migration algorithm that uses these messages.

7.2.1 The migration algorithm

Assume that we need to move an element E_a from P_i to P_j, that is $E_a \in \Pi_i^{t-1}$ and $E_a \in \Pi_j^t$. Assume also $Adj(E_a) = \{V_{p_1}, \ldots, V_{p_n}\}$ is the set of vertices of E_a. We initially send a *Move-Msg*$(i,j) = \{E_a\}$ message from P_i to P_j. If $V_p \in Adj(E_a)$ and also $(P_j, V_p^j) \in Ref(V_p^i)$ we only include the reference in the message. Otherwise if $V_p \in Adj(E_a)$ and $(P_j, V_p^j) \notin Ref(V_p^i)$ we include the node V_p in the message. P_j creates the copy V_p^j and it initializes $Ref(V_p^j) = Ref(V_p^i) \cup (P_i, V_p^i)$. At this point P_j has a reference to all the copies of V_p. It then sends an *AddRef-Msg*$(j,i) = \{(V_p^i, V_p^j)\}$ for each reference in $Ref(V_p^j)$ and all the other copies update their references to the new copy. Using V_p^j processor P_j creates the element E_a. P_i then deletes its element E_a. It can happen that E_a was the only element that pointed to V_p in P_i. In this case we wish to remove also V_p^i. P_i sends a *DelRef-Msg*$(i,j) = \{(V_p^j, V_p^i)\}$ to each reference in $Ref(V_p^i)$. Once all the other processors remove their references to V_p^i we can delete the node. A simple illustration of this algorithm is shown in Figure 21.

In this example we show a mesh with two elements partitioned between three processors P_0, P_1 and P_2. Our goal is to move the element E_a form P_0 to P_2. We initially send a *Move-Msg*$(0,2) = \{E_a\}$ that includes V_0, V_1 and V_2. We initialize $Ref(V_0^2) = \{(P_1, V_0^1), (P_0, V_0^0)\}$. We similarly initialize $Ref(V_1^2)$ and also $Ref(V_2^2)$. P_2 then sends a *AddRef-Msg* to P_0 and P_1 to the copies of V_0, V_1 and V_2 that includes references to V_0^2, V_1^2 and V_2^2. P_0 then deletes its copy of E_a. Since it can also remove V_0^0, V_1^0 and V_2^0, it sends a *DelRef-Msg* to the other processors.

We will explain the algorithm in more detail using the example in Figure 22. There we show a mesh composed of 8 elements $(E_a, \ldots, E_h)$ and 9 nodes $(V_0, \ldots, V_8)$ partitioned between 4 processors $(P_0, \ldots, P_3)$. In the top Figure we show the initial partition Π^{t-1} and in the bottom Figure we show the target partition Π^t. The initial representation of the mesh is shown in Figure 23 (a). Our goal is to move the elements from the initial partition to the destination partition. This can be done by executing the commands:

- P_0: move E_a to P_3 by sending the message *Move-Msg*$(0,3) = \{E_a\}$.

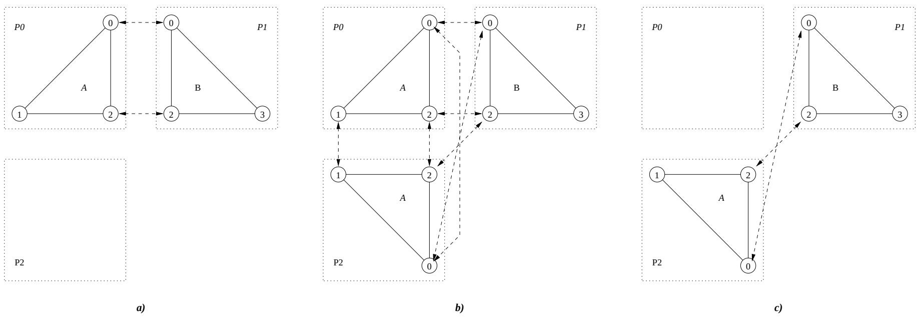
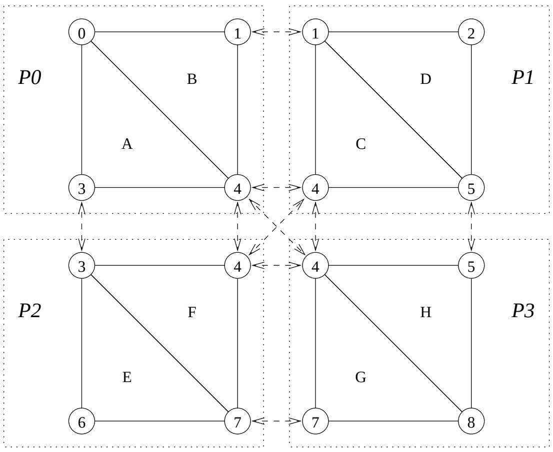


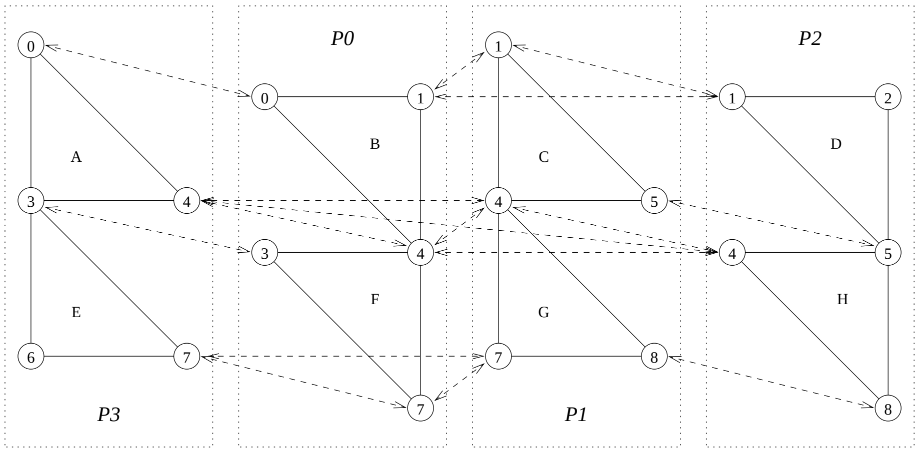
Figure 21: A migration example. (a) shows the initial mesh. The goal is to move E_a from P_0 to P_2. We first copy E_a to P_2 (b) and then we delete the element E_a in P_0 (c).

- P_1: move E_d to P_2 by sending the message $\mathit{Move\text{-}Msg}(0,2) = \{E_d\}$.
- P_2: move E_e to P_3 and E_f to P_0 by sending the messages $\mathit{Move\text{-}Msg}(2,3) = \{E_e\}$ and $\mathit{Move\text{-}Msg}(2,0) = \{E_f\}$.
- P_3: move E_g to P_1 and E_h to P_2 by sending the messages $\mathit{Move\text{-}Msg}(3,1) = \{E_g\}$ and $\mathit{Move\text{-}Msg}(3,2) = \{E_h\}$.

- First we send the elements to the destination processor. If there is an element E_a located in P_i and $E_a \in \Pi_j^t$ then we send a $\mathit{Move\text{-}Msg}(i,j) = \{E_a\}$ message from P_i to P_j. If an element E_a refers to a node V_p^i of which P_j has no local copy then P_i must also include the node in the message. Determining if P_j has a local copy of V_p is easy: we only need to look at the references to remote copies of V_p^i (is $(P_j, V_p^j) \in \mathit{Ref}(V_p^i)$?) in the sending processor P_i. If we find that P_j has a local copy V_p^j then we use that copy to create the element E_a in P_j. When we send a node we also include all the references to other copies. This way the receiving processor can create its local copy and then send a message to the other processors to update their references to it. Also when we are sending multiple elements to a processor we need to be careful to include only one copy of the nodes. The description of this phase is shown in Figure 24. The initial messages for the previous example are:
  - P_0: move E_a to P_3 by sending the message $\mathit{Move\text{-}Msg}(0,3) = \{E_a\}$. Include in the message the nodes V_0 and V_3 and a reference to V_4^3. In P_3 use these two nodes and the existing copy of V_4 to create the element E_a.
  - P_1: send E_d to P_2 by sending the message $\mathit{Move\text{-}Msg}(1,2) = \{E_d\}$ with the nodes V_1, V_2 and V_5.



a)



b)

Figure 22: Migration of elements from an initial partition Π^{t-1} (a) to a target partition Π^{t} (b).

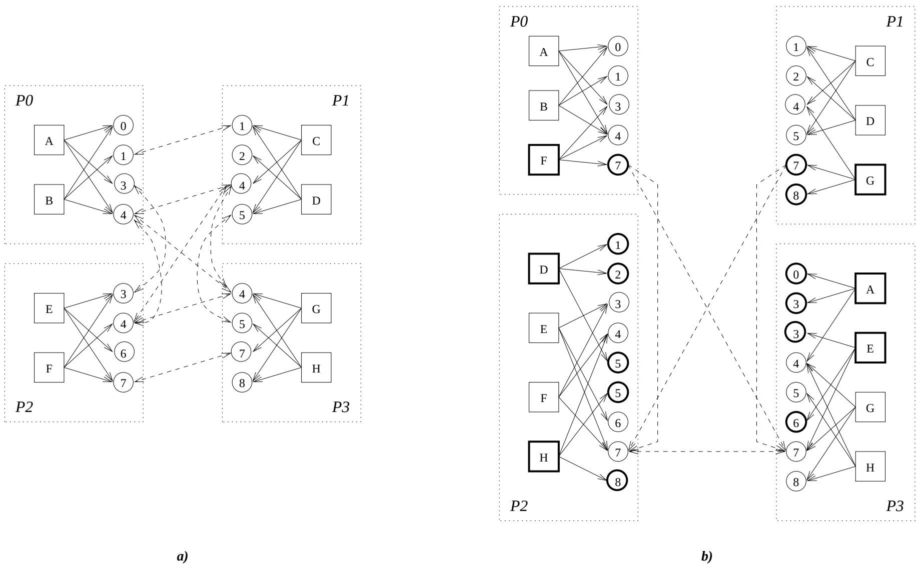


Figure 23: A migration example: internal representation of the mesh at the beginning of the migration (a) and after copying the elements to the destination processors (b).

```
FOR each element E_a such that E_a ∈ Π_i^{t-1}, E_a ∈ Π_j^t, i ≠ j  DO
 insert E_a into Move-Msg(i, j).
  FOR each node V_p ∈ Adj(E_a)  DO
    IF (P_j, V_p^j) ∈ Ref(V_p^i)  THEN
     insert (P_j, V_p^j) into Move-Msg(i, j).nodes(E_a).
    ELSE
     insert V_p into Move-Msg(i, j).nodes(E_a).
     insert (P_i, V_p^i) into Move-Msg(i, j).ref(V_p).
      FOR each reference (P_k, V_p^k) ∈ Ref(V_p^i)  DO
       insert (P_k, V_p^k) into Move-Msg(i, j).ref(V_p).
      END FOR
    END IF
  END FOR
END FOR
FOR each processor P_j  DO
  IF i ≠ j and Move-Msg(i, j) ≠ ∅  THEN
   send Move-Msg(i, j).
  END IF
END FOR
```

Figure 24: Migration phase: sending the elements to the destination processors.

  - P_2: similarly send E_e to P_3 (with V_3 and V_6 and a reference to V_7^3) and E_f to P_0 (with V_7 and reference to V_3^0 and V_4^0).
  - P_3: at the same time send E_g to P_1 (with V_7 and V_8 and a reference to V_1^4) and send E_h to P_2 (with V_5 and V_8 with a reference to V_2^4).

- Once a processor P_j receives a message $Move\text{-}Msg(i, j)$ it first creates the new nodes as specified in the message and then constructs the elements. If V_p is a node of a an element E_a such that $E_a \in Move\text{-}Msg(i, j)$ and $V_p \in Move\text{-}Msg(i, j).nodes(E_a)$ then a new copy V_p^j is created in P_j and $Ref(V_p^j)$ is initialized to $Move\text{-}Msg(i, j).ref(V_p)$ (remember that this also includes a reference to the sending processor (P_i, V_p^i)). At this point $(P_i, V_p^i) \in Ref(V_p^j)$ but $(P_j, V_p^j) \notin Ref(V_p^j)$. It is responsibility of P_j to inform the other processors of the newly created copy.

  Continuing with the example, when P_2 sends E_f to P_0 it should also include a copy of V_7. Before P_1 constructs the element E_f it should first create the node V_7^0. Using that node and the local copies V_3^0 and V_4^0 it can then create the element E_f. Note that the copy of V_7 in P_0 has a reference to the copies in P_2 and P_3 and not vice versa. It is the responsibility of P_0 to inform the other processors of the newly created copy. When P_1 receives the element E_g it also creates a copy of V_7 but the copies in P_0

```
FOR each message Move-Msg(i, j) sent from other processor P_i to P_j  DO
 receive Move-Msg(i, j).
  FOR each element E_a ∈Move-Msg(i, j)  DO
    FOR each node V_p ∈ Move-Msg(i, j).nodes(E_a)  DO
      IF V_p does not exist in P_i  THEN
        create the node V_p and initialize Ref(V_p^i) = Move-Msg(i, j).ref(V_p).
      END IF
    END FOR
   construct the element E_a.
  END FOR
END FOR
```

Figure 25: Migration phase: creating the elements in the destination processors.

and P_1 of that node know nothing about each other at this moment.

There is another problem: P_2 receives E_d from P_1 and E_h from P_3 and both messages include the vertex V_5 (a similar problem happens in P_3 with V_3). We will explain later how to handle these conditions. Figure 23 (b) shows the mesh at this stage and Figure 25 presents an outline of this phase.

- In the next phase we update the references to the new nodes. Assume that V_p^i is a node created in P_i in the previous phase as the result of a $MoveMsg(j, i)$. P_i needs to inform P_j and all the other processors P_k that have a copy of V_p about the location of V_p^i in memory so they can create a reference to it. Using $Ref(V_p^i)$, P_i sends a $AddRef\text{-}Msg(i, k)$ for each reference $(P_k, V_p^k) \in Ref(V_p^i)$. This procedure is shown in Figure 26.

  In the previous example P_0 sends a message to P_2 and P_3 to update their references to V_7^0 and so does P_1. P_2 detects that there is more than one new copy of V_7 so it informs P_0 to update the reference from V_7^0 to V_7^1. The same thing happens in P_3.

  At this stage we detect that there are two copies of V_5 in P_2 and two copies of V_3 in P_3. One of the copies is destroyed and the corresponding elements are updated accordingly. The state of the mesh at the end of this phase is shown in Figure 27 (a) and (b).

- We now remove the elements from the source processors. Moreover if there is some node V_p such that $ElemAdj(V_p) = \varnothing$ we delete the node to free memory. In the example the destruction of E_e and E_f in processor P_2 causes the deletion of V_3^2, V_6^2 and V_7^2. Note that the node V_4^2 is referenced by the new element E_h so it is not deleted.

  Before deleting the nodes we send a $DelRef\text{-}Msg$ message to the processors that have a reference to the node indicating that they should remove the

FOR each new node V_p^i **DO**
 FOR each reference $(P_j, V_p^j) \in$ *Ref*(V_p^i) **DO**
  insert (V_p^j, V_p^j) into *AddRef-Msg*(i, k).
 END FOR
END FOR
FOR each processor P_j **DO**
 IF $i \neq j$ and *AddRef-Msg*$(i, j) \neq \varnothing$ **THEN**
  send *AddRef-Msg*(i, j).
 END IF
END FOR
FOR each message *AddRef-Msg*(j, i) sent from other processor P_j to P_i **DO**
 receive *AddRef-Msg*(j, i).
 FOR each reference $(V_p^i, V_p^j) \in$ *AddRef-Msg*(j, i) **DO**
  insert (P_j, V_p^j) into *Ref*(V_p^i).
 END FOR
END FOR

Figure 26: Migration phase: updating the references to the new nodes.

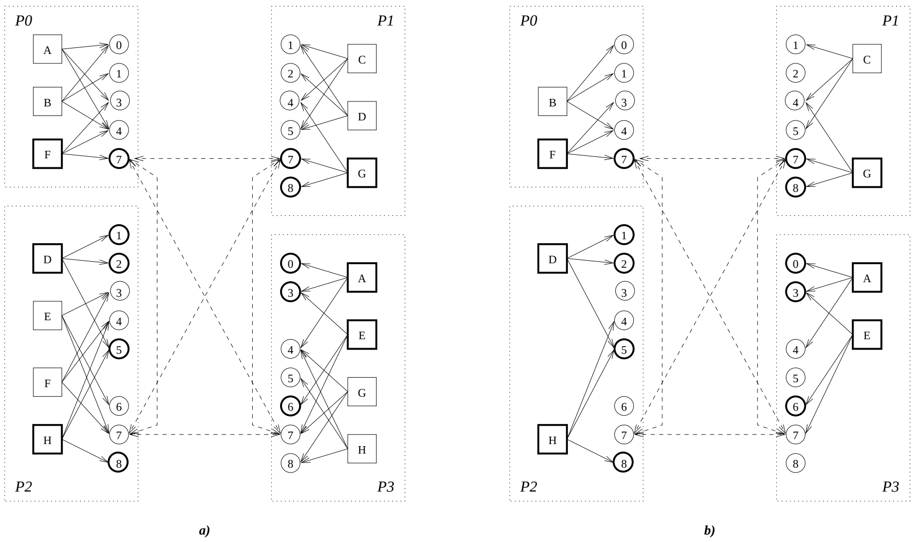


Figure 27: A migration example: internal representation of the mesh after copying the elements to the destination processors (a) and after removing the elements from the source processors (b).

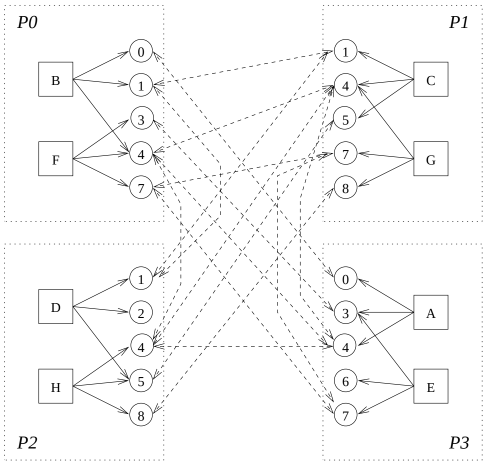


Figure 28: A migration example: internal representation of the mesh at the end of the migration phase.

reference. Finally we destroy the nodes. The representation of the mesh at the end of the migration is shown in Figure 28. Figure 29 shows this procedure.

8 Project overview

In finite element programs written in FORTRAN meshes are typically represented by a table of elements and a table of nodes. Each row in the table of nodes corresponds to a node and in its columns we store the coordinates of the node. The elements are stored in the table of elements and each element keeps track of its vertices by storing their indices in the table of nodes.

This storage scheme allows for compact representations. This is a desirable property as real problems have thousands of elements and nodes. Also it is very easy to find the nodes of an element and the coordinates of a node. These are the two most common operations that are required to construct the local and global matrices explained in Section 5. Unfortunately the storage scheme using simple arrays is inflexible for a dynamic environment where the nodes and elements are continuously created and deleted as the result of the refinement and coarsening algorithms.

In this section we will give an overview of the object-oriented approach that we have taken to support the dynamic mesh adaptation. We will also explain how we implemented the remote references as *smart pointers* to avoid memory leaks or dangling references.

Our program has been designed to run on distributed memory parallel computers. The current version runs in parallel in a network of SUN workstations. For communication we use the MPI [23] message passing library. In particular

```
FOR each element E_a such that E_a ∈ Π_i^{t-1}, E_a ∈ Π_j^t, i ≠ j  DO
 delete element E_a.
  FOR each node V_p ∈ Adj(E_a)  DO
   remove E_a from ElemAdj(V_p).
    IF ElemAdj(V_p) = ∅  THEN
      FOR each reference (P_j, V_p^j) ∈ Ref(V_p^i)  DO
       insert (V_p^j, V_p^i) into DelRef-Msg(i, j).
      END FOR
     delete V_p^i.
    END IF
  END FOR
END FOR
FOR each processor P_j  DO
  IF i ≠ j and DelRef-Msg(i, j) ≠ ∅  THEN
   send DelRef-Msg(i, j).
  END IF
END FOR
FOR each message DelRef-Msg(j, i) sent from other processor P_j to P_i  DO
 receive DelRef-Msg(j, i)
  FOR each reference (V_p^i, V_p^j) ∈ DelRef-Msg(j, i)  DO
   remove (P_j, V_p^j) from Ref(V_p^i).
  END FOR
END FOR
```

Figure 29: Migration phase: deleting the elements on the source processors.

our program uses the MPICH [26] implementation of MPI from the Argonne National Lab. MPI is becoming the standard for message passing libraries and there are efficient implementations for many parallel computers. Although MPI has many different ways of sending a message between two computers we use the standard blocking send and receive. When a processor P_i wants to send a memory buffer to another processor P_j it calls a C function and blocks until it is safe to reuse the buffer. This does not guarantee that the message is actually delivered. When a processor P_j wants to receive a buffer from P_i it calls another C function and waits until a message from P_i arrives to P_j. We have designed C++ wrappers around these C routines. In our environment a message is just another kind of object.

We also designed a very simple window interface that allows us to select an element or a group of elements for refinement. We use windows to display the mesh where the elements are colored depending on which processor they are located. P_0 is usually responsible for managing the windows. This processor collects information from all the other processors and broadcasts the user commands to them.

Finally we use the *Chaco* [20] graph partitioning program to generate the initial partitions and for some of the mesh repartitioning algorithms. Since *Chaco* is a sequential program we also run it in P_0.

8.1 The user interface

The user interface is designed around the Tcl/Tk scripting package [28], [29]. The user is presented with a window that displays the mesh. Using the mouse the user can click on individual elements to select them or it can drag the cursor to select a group of elements. He can then choose commands from the menus to refine, partition or migrate the mesh using different algorithms. There are several options to display information about a mesh or about individual elements. The user can load different meshes using a file selection box and can select different initial partition files for a particular mesh. There is also an option for zooming regions of the mesh. A sample display with the mesh used in Section 7.1 is shown in Figure 30. The elements are colored according to their processor assignment. In this example the mesh is partitioned between 4 processors.

Although the window is managed by P_0 the actual elements and nodes are located in different processors. When we want to display the mesh all the processors need to send a message to P_0 that contains the elements and its coordinates. When the user issues a command by selecting an option from the menus P_0 broadcasts the command to all the other processors. All this distributed input and output is managed by 4 classes. When the user selects an option from the menu the program calls a method of a `Console` object located in P_0. This object is responsible for putting a wrapper around the command and broadcasting it to all the other P_i processors. When P_i receives a command message from P_0 it invokes a method on its `ConsoleStub` object. This object unwraps the command and calls the appropriate method on the

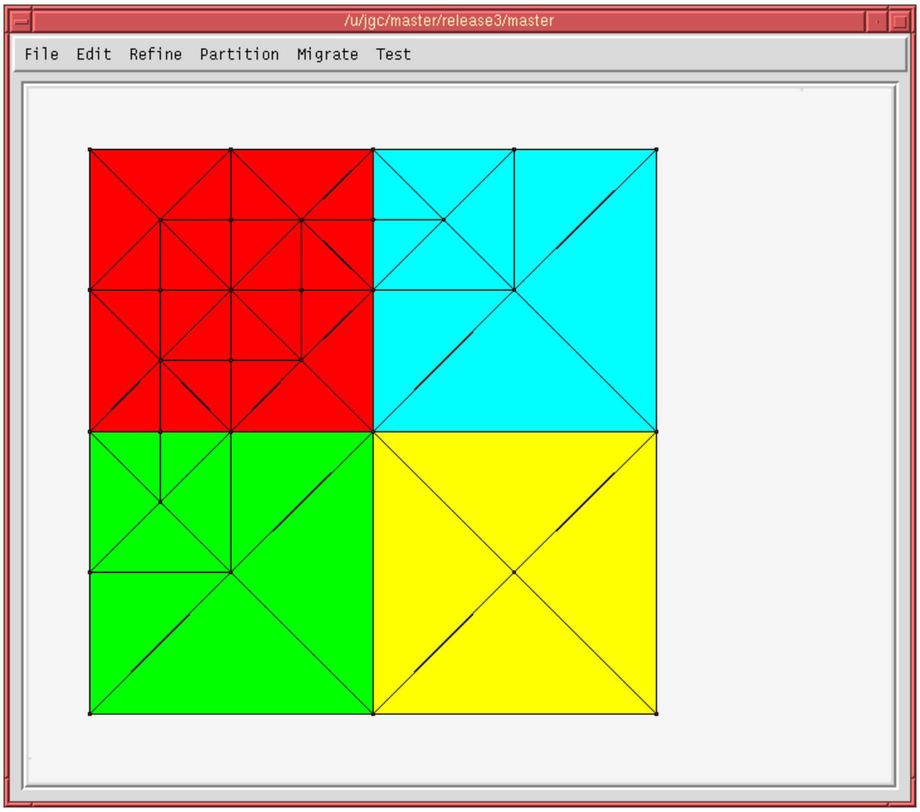


Figure 30: Sample display of the window interface of the program.

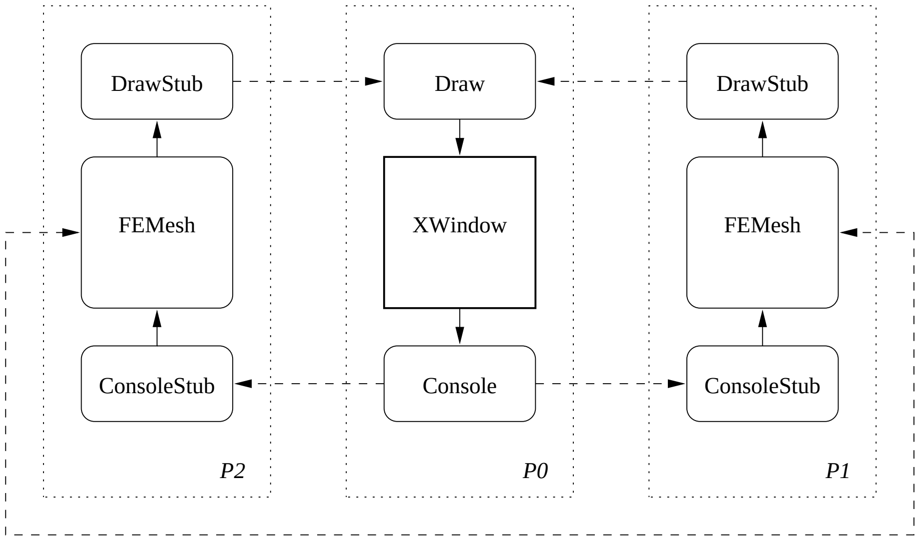


Figure 31: Implementation of the distributed input/output.

`FEMesh` object that implements the mesh. This `FEMesh` object executes the command, probably communicating with some other `FEMesh` objects in other processors. When P_i wants to produce some output like drawing an element in the screen it calls a method on its `DrawStub` object. This object collects several output instructions and sends an individual message to P_0. P_0 receives the output messages through the `Draw` object. Finally to display the output the `Draw` object calls the appropriate Tcl/Tk commands. The relations between these objects are shown in Figure 31.

8.2 Object oriented representation of the FE mesh

The distributed mesh is implemented using the `FEMesh` class. There is only one `FEMesh` object per processor. As this class is in essence a container for elements and nodes its two most important data members are a list of pointer to elements and a list of pointers to nodes. These lists are implemented using the Tools++ [30] templates library. The list of elements is a list of pointers to the elements in the fine mesh assigned to the processor while the list of nodes is a list of pointers to all the nodes in the processor. The `FEMesh` class has also a pointer to a root element. The children of this distinguished element are the elements of the coarse mesh assigned to the processor. In this way it is possible to traverse down in the element hierarchy from the elements in the coarse mesh to their descendants in the fine mesh. The `FEMesh` class has methods for all the algorithms described in the previous sections.

The remote references described in Section 5.2 are implemented using the

NodeRef class. This class is just a pair consisting of a processor and a memory address. It represents a reference to a node located in that processor at a specific memory location. To invoke a method into a node located in a remote processor these addresses are packed into a message as long integers (MPI does not support the notion of messages of pointers) and sent to the remote processor. Once the message arrives at the destination processor the address is cast into a pointer to the node and the corresponding method is invoked on the node. For this reason it is very unsafe to delete or move a node if the other processors still have a reference to it. This restriction had a big influence on the migration algorithm of Section 7.2.

The elements are implemented using inheritance. There is an abstract class AbsElement from where we derive classes for the different types of elements such as Triangle and Quadrilateral. An element has a pointer to its parent, that is the element whose refinement created it. If the element was also refined it has a list of pointers to its children. Finally the element has a vector of pointers to its vertices. This makes it easy to access the coordinates of the vertices of the element to generate its local matrices for the numerical simulations.

Another important class is the Node class. This class keeps track of the coordinates of the node and has a list of NodeRef to the copies of the node in other processors. For an internal node this list is empty. For a node located in an internal boundary between processors this list contains the references to the copies of the node in the remote processor. The Node class has also a list of pointers to their adjacent elements. When a element is created it is automatically added to the lists in its vertices. When the element is deleted it is removed from the lists of its vertices. If the list of pointers to the adjacent elements of a Node object becomes empty then there is no element pointing to that node. In this case the node can be safely deleted without leaving any dangling pointers to it.

The messages are also encapsulated in classes that inherit from a common Message class. To send or receive a message the user just calls the send and receive methods of the message object. These classes also handle all the manipulation of the buffers so the user does not have to call MPI routines directly. The MPI functions that are not related to messages, such as obtaining the processor number or the number of processors, are encapsulated in an MPI class.

8.3 File format

The file format for the meshes is very simple. Each mesh description consists of a header line that includes the number of nodes and the number of elements. After this header line there is a line for each node that includes the node number and its coordinates. After all the nodes there is a line for each element. For each element we include its type, the element number and the number of its vertices.

9 Experimental results

To evaluate the quality and performance of our system we performed a series of tests on a network of SUN SparcStation10 workstations, each with 32MB of RAM and running Solaris 2.4. The processor P_0 that was responsible for the window interface and the serial part of the refinement algorithm is a multiprocessor workstation with 64MB. We tested our program using between 4 and 32 processors. These machines were scattered between the 1st and the 5th floor of the CIT building and were connected by a 10Mbs ethernet network. Most of the machines that we used were located in the SunLab in the 1st floor while P_0 was located in the 5th floor. The network is divided in several subnetworks that were connected through gateways in the 5th floor. In particular, to send a message from a machine in one row of the SunLab to a machine in a different row (that is connected to a different subnetwork) the message has to travel all the way up to the 5th floor and then all the way down to the SunLab.

The tests cover the major components of the system. We found that the cost of the refinement algorithm is dominated by the serial part. By performing a sequence of successive refinements of the whole mesh we obtained some very big meshes. Our parallel Parallel Nested Repartition algorithm computed high quality partitions in a very reasonable time. By using the refinement history we were able to obtain better partitions than in other multilevel algorithms.

We ran these test when most of the machines were idle but there is no guarantee that the timings are not influenced by other users.

9.1 Network performance

The first test does not evaluate our system directly. Instead our goal is to determine the performance of the network and compare it with a real parallel computer. We tried three sets of programs on machines located in the 5th floor and machines located in the 1st floor. The intuition was that as the distance between machines in the SunLab is longer than the distance between the machines in the 5th floor their messages should take more time.

The first test is a point-to-point communication program where a processor P_i sends a message to some other processor P_j and waits for a response. We tried this test for several messages whose length ranged from 1 to 100000 `double`. Table 1 shows the results of this test measured in MBytes per seconds. These results are plotted in Figure 32 (a). Sending a message of only 1 `double` takes 0.0015 sec. (this is the latency of the message) while the cost of sending an additional byte for long messages is 0.0015 msec and the maximum performance was around 1 MByte/sec., consistent with performance obtainable a using 10Mbit/sec ethernet network. We did not experience too much difference between machines in the lab and machines connected to the same subnetwork. In both cases only when the messages are around 8K we do obtain full speed.

The second test corresponds to the broadcast operation. In this case a distinguished processor P_0 sends a different message to all the other processors and waits for a response from all of them. This operation is usually executed

Length	5th floor	SunLab
1	0.005234	0.004999
5	0.025177	0.021793
10	0.050197	0.034597
50	0.179837	0.166160
100	0.260687	0.246555
500	0.586020	0.565639
1000	0.741308	0.725855
5000	0.786961	0.766936
10000	0.820302	0.764463
50000	0.843152	0.820467
100000	0.859652	0.838322

Table 1: Point to point data rates between machines located in the 5th floor and machines located in the SunLab. Performance is measured in MBytes/sec and the length of the message is in `double`.

when P_0 broadcasts the result of the repartitioning algorithm. The results for 4, 8 and 16 processors are shown in Table 2 and plotted in Figure 32 (b), (c) and (d). Again the full speed is obtained when the messages are of 8K. For longer messages we start to notice a degradation of the performance, possible due to contention. In these tests the maximum performance was also around 1 MByte/sec.

	4 Processors		8 Processors		16 Processors	
Length	5th floor	SunLab	5th floor	SunLab	5th floor	SunLab
1	0.007748	0.006733	0.006648	0.004730	0.005601	0.006167
5	0.038316	0.033345	0.030359	0.025214	0.026274	0.027965
10	0.077707	0.066563	0.060923	0.049014	0.051953	0.057124
50	0.329701	0.300788	0.326305	0.158403	0.245903	0.282621
100	0.515726	0.494174	0.531610	0.491153	0.465172	0.517887
500	0.838217	0.896251	0.952069	0.972119	0.917585	1.041698
1000	0.937303	0.998263	1.026578	0.958222	1.048212	1.046217
5000	0.819534	0.861735	0.819441	0.861253	0.931990	0.908919
10000	0.757282	0.746826	0.915833	0.869889	0.936937	0.917223
50000	0.849323	0.809480	0.891440	0.842223	0.904827	0.842918
100000	0.862556	0.833068	0.881041	0.842355	0.850441	0.837908

Table 2: Broadcast from a single source to all the other processors. Performance is measured in MBytes/sec and the length of the message is in doubles.

The last test is an all-to-all communication program. In this test each processor sends a message to each other processor and waits for a response. This test is an extreme case of our migration algorithm. The effect is that it saturates

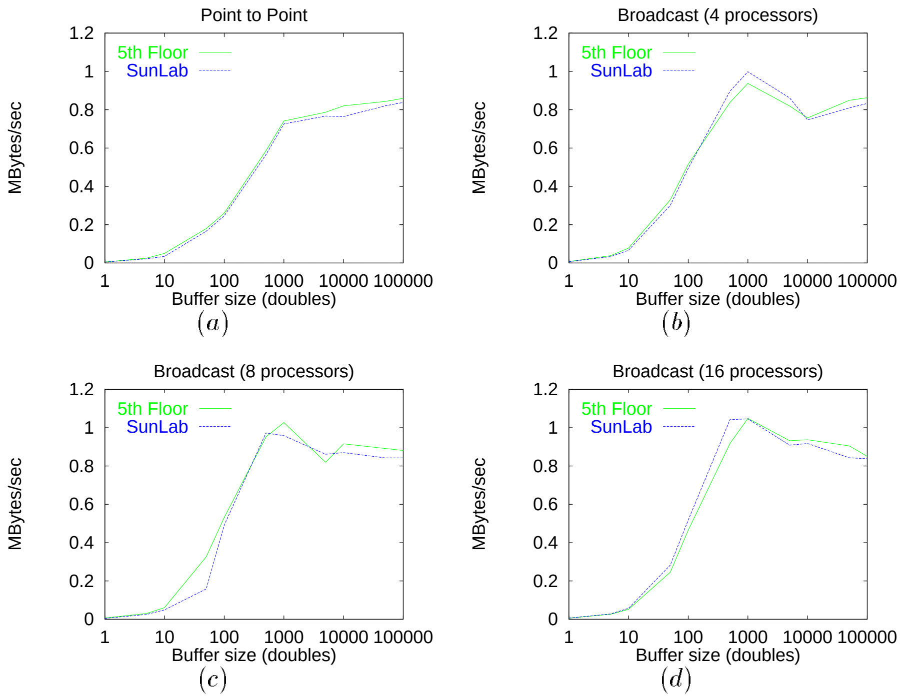


Figure 32: Comparison of the performance of the network between machines in the 5th floor and machines in the SunLab. (a) shows the performance of a ping application where a source processor sends a message to a destination processor and then waits for a response. This example is used to measure the latency of the network. Only when the messages are around 8K (1000 doubles) we do obtain full speed. (b), (c) and (d) show the performance of a broadcast example where a source processor sends an individual message to every other processor and waits for a response from all of them.

the network as it is shown in Table 3 and Figures 33 (*a*), (*b*) and (*c*). As it is shown in the figures when we increase the number of processors we notice a lower performance due to contention. The maximum speed for the 4 processor case was around 0.6 MByte/sec., almost half of the performance of the point to point program. When we increase the number of processors to 16 the maximum performance drops to 0.2 MByte/sec. (compared with 1 MByte/sec. in the point to point test).

	4 Proc		**8 Proc**		**16 Proc**	
Length	**5th floor**	**SunLab**	**5th floor**	**SunLab**	**5th floor**	**SunLab**
1	0.002787	0.003874	0.002790	0.002305	0.000545	0.000169
5	0.041049	0.014917	0.018661	0.015755	0.010178	0.007148
10	0.075899	0.033313	0.066663	0.028382	0.024450	0.015553
50	0.360098	0.276849	0.194468	0.073495	0.158196	0.059094
100	0.483465	0.248942	0.251540	0.091595	0.196801	0.105000
500	0.490462	0.566953	0.313826	0.265786	0.195512	0.147489
1000	0.550117	0.363984	0.315014	0.258172	0.224284	0.184256
5000	0.435300	0.558364	0.329452	0.317619	0.238315	0.187827
10000	0.489746	0.559008	0.318954	0.340022	0.222170	0.195876
50000	0.496505	0.537064	0.316744	0.346304	0.116656	0.196713
100000	0.504589	0.531281	0.201673	0.324911	0.078395	0.139036

Table 3: All to all communication. Performance is measured in MBytes/sec and the length of the message is in doubles.

Finally we ran the same tests in an IBM SP-2 with 24 processors. These results are shown in Table 4 and plotted in Figure 33 (*d*). The SP-2 was able to run the same tests around 35 times faster than the network of workstations. Furthermore contention has a much smaller effect on that machine.

9.2 Mesh examples

Our tests were run on two basic meshes. The first mesh is of relatively small size. It contains 9000 triangular elements and 4720 nodes and is a 2D unstructured FE grid of an airfoil. This mesh is provided with the *Chaco* program and it is known as the Hammond mesh. In our examples we will refer to it as `air.mesh`. Several views of this mesh are shown in Figure 34.

The second mesh is a larger 2D finite element grid of around 30000 triangular elements and 15000 nodes. We will refer to it as the `big.mesh`. Four different views of this mesh are displayed in Figure 35. As it is shown in these pictures there is a big disparity on the size of the elements.

9.3 Initial partition of the mesh

Recall that the first task of the general algorithm for computing the solution of dynamic systems in Figure 7 was to obtain an initial partition of the mesh. This

Length	pt2pt	bcast 4	bcast 8	all2all 4	all2all 8
1	0.030493	0.049504	0.032990	0.049397	0.047356
5	0.449846	0.936888	0.728459	0.820869	0.786651
10	0.962093	1.980803	1.431854	1.617866	1.552719
50	3.441315	6.545091	5.087394	6.464023	6.080583
100	5.241769	10.247676	8.709354	10.275279	9.726054
500	14.986043	21.422860	20.492599	21.749445	22.099546
1000	19.998413	24.756026	26.153441	27.523068	25.587889
5000	28.619531	27.910036	22.897113	31.495189	22.869594
10000	31.171557	32.700909	24.209284	31.805331	22.370358
50000	33.140733	34.230389	25.071044	33.086489	24.191446
100000	32.841156	32.861832	25.181348	32.626888	25.082283

Table 4: Point to point, broadcast and all to all communication on the SP2. Performance is measured in MBytes/sec. and the length of the message is in doubles.

partition is usually computed using a serial computer during a preprocessing step and it is not part of our system. Nevertheless it allows us to compare the quality of the partitions obtained using serial multilevel algorithms with more standard algorithms like Recursive Spectral Bisection [15]. Recursive Spectral Bisection is known to produce very good partitions but it is too expensive to use for repartitioning the mesh.

Our program assumes that there is an initial partition of the mesh and we generate this partition using *Chaco* in a preprocessing step. This system provides several method for partitioning a graph. We used both Recursive Spectral Bisection and Multilevel Bisection. As *Chaco* is a serial program we run these tests on a single SparcStation10 workstation with 64MB of RAM. The results are shown in Tables 5, 6, 7 and 8. The time required to compute the partitions of both meshes is shown in Figure 36 and the number of shared nodes is shown in Figure 37. Clock time is the time elapsed to compute the partition while user and system time denote the time spent in user and system mode. The difference between clock time and the sum of user and system time represents the time the system was idle because of trashing. Remember that this partition is computed using the dual of the mesh. In this case the row labeled "edges cut" is the number of edges cut in the dual of the mesh. Average elements is the number of elements in each processor while shared nodes is the number of nodes in the internal boundaries between processors. A lower number of shared nodes represents a better partition as it requires less communication.

In these examples *Chaco*'s Multilevel Bisection outperformed *Chaco*'s Spectral Bisection both in time and in the quality of the partitions. The low performance of RSB on computing the partitions for the `big.mesh` was due to the fact that it required a considerable amount of memory, more than the 64MB available in the computer. Although all the partitions required more than 4:30

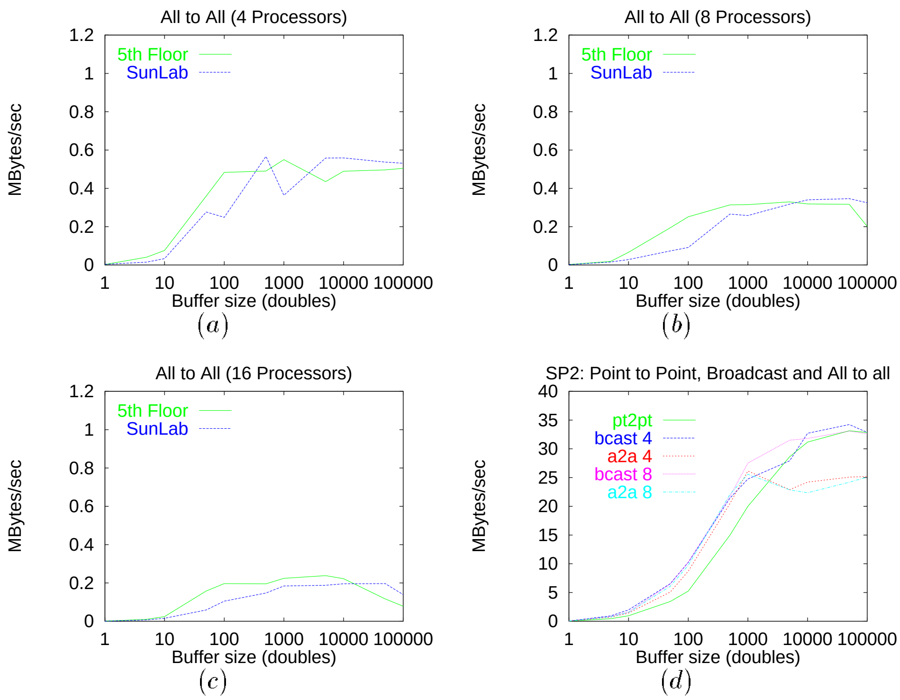


Figure 33: Comparison of the performance of the network between machines in the 5th floor and machines in the SunLab and the performance of the same applications run on the SP2. (a), (b), (c) show the performance of an all to all communication example where each processor sends an message to each other. These example are a measure of the contention on the network. (d) is the result of running the same tests on the SP2.

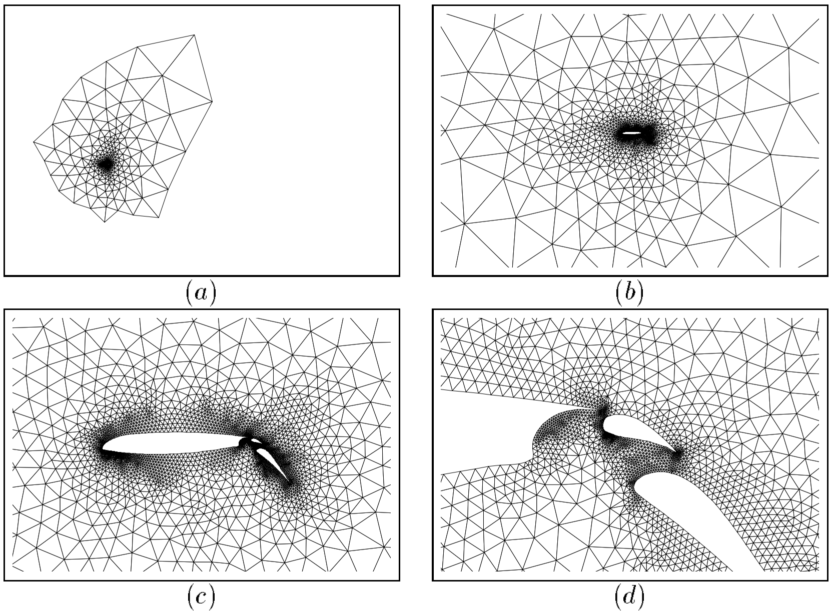
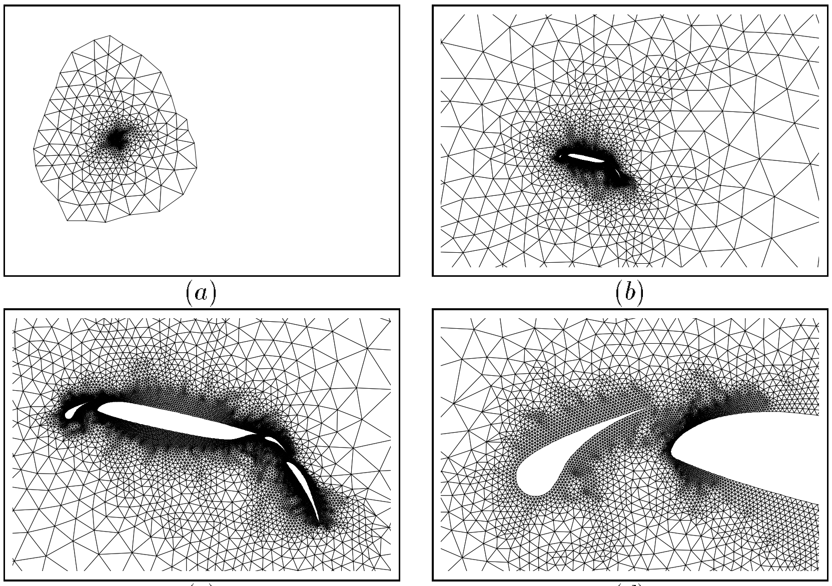


Figure 34: The `air.mesh` is a 2D finite element grid of a complex airfoil with triangular elements. It consists of 9000 elements and 4720 nodes. This mesh is provided with the *Chaco* program.



(a) (b) (c) (d)

Figure 35: The `big.mesh` is a 2D finite element mesh of a complex airfoil. It consists of 30269 elements and 15606 nodes. It is obtained from riacs.edu in the directory "pub/grids/big.*".

hours of clock time only 1 hour was spent doing useful work. In all cases the serial Multilevel Bisection algorithm produced better partitions in less than a minute.

	Number of partitions			
	4	**8**	**16**	**32**
Clock Time	4:54.8	5:57.1	6:49.4	6:48.5
User Time	4:49:0	5:51.2	6:40.9	6:36.8
System Time	3.7	5.4	8.0	11.1
Edge cuts	928	1702	2747	4417
Avg. elements	2250	1125	563	281
Shared nodes	144	267	428	690

Table 5: Spectral Bisection on the `air.mesh` using *Chaco* on a 64Mb Sun SparcStation. The dual of the mesh has 9000 vertices and 52507 edges. The times are in hours:minutes:seconds. Edge cuts is the number of edges cut reported by Chaco.

	Number of partitions			
	4	**8**	**16**	**32**
Clock Time	5:24:43.7	4:16:16.6	4:43:06.6	4:34:39.7
User Time	42:45.5	49:29.9	1:00:13.0	1:02:58.5
System Time	18:53.6	19:35.0	19:27.2	19.57.6
Edge cuts	1929	3252	5427	8084
Avg. elements	7567	3784	1892	946
Shared nodes	298	494	834	1243

Table 6: Spectral Bisection on the `big.mesh` using *Chaco* on a 64Mb Sun SparcStation. The dual of the mesh has 30269 vertices and 178639 edges. The times are in hours:minutes:seconds.

9.4 Refinement of the mesh

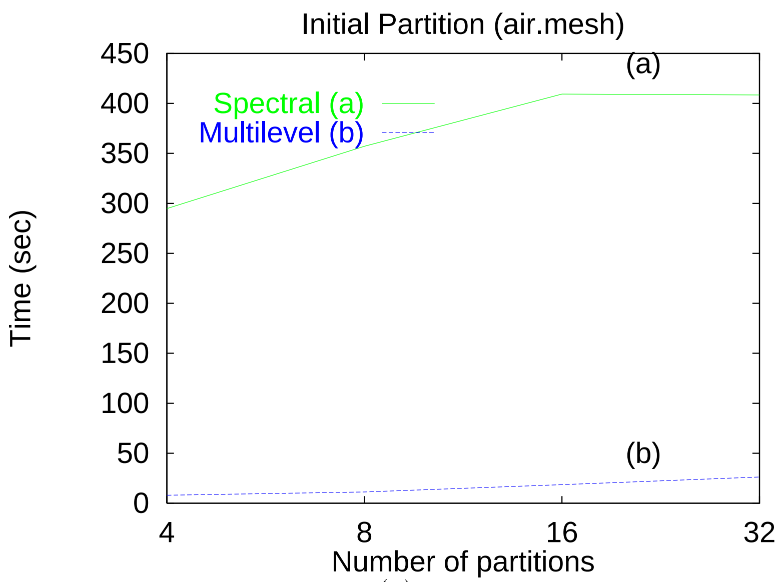
To test the refinement algorithm we performed successive refinements of the mesh. In each of these phases all the elements of the mesh are selected for refinement. The number of elements grows exponentially with the level of refinement. By doing a series of successive refinements we were able to create meshes consisting of more than 2,000,000 elements. We estimate that we could store around 40,000 to 50,000 elements in each processor. After that the data sets were too big and the performance of the algorithm drops considerably due to trashing. These results are shown in Tables 9, 10, 11 and 12. The serial time is the time spent creating new elements and nodes and the communication time is the time spent propagating the refinement to adjacent processors. For each

	Number of partitions			
	4	**8**	**16**	**32**
Clock Time	8.0	11.2	18.6	26.2
User Time	6.4	9.7	15.2	21.1
System Time	0.6	1.3	3.2	4.9
Edge cuts	878	1510	2440	3978
Avg. elements	2250	1125	563	281
Shared nodes	127	238	371	613

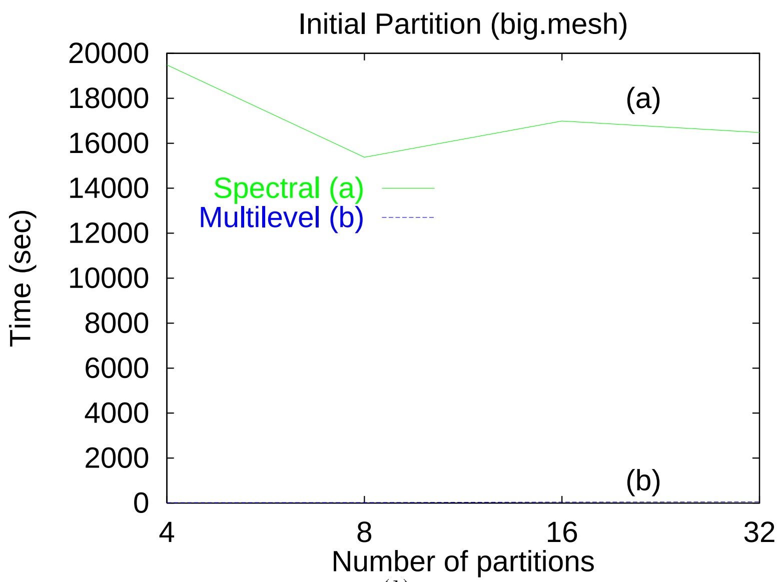
Table 7: Serial Multilevel Bisection on the `air.mesh` using *Chaco* on a 64Mb Sun SparcStation. The times are in hours:minutes:seconds.

	Number of partitions			
	4	**8**	**16**	**32**
Clock Time	17.9	23.4	47.4	52.7
User Time	16.3	21.4	43.0	46.3
System Time	1.3	1.8	4.1	6.1
Edge cuts	1575	2509	4047	6701
Avg. elements	7567	3784	1892	946
Shared nodes	233	374	619	1026

Table 8: Serial Multilevel Bisection on the `big.mesh` using *Chaco* on a 64Mb Sun SparcStation. The times are in hours:minutes:seconds.

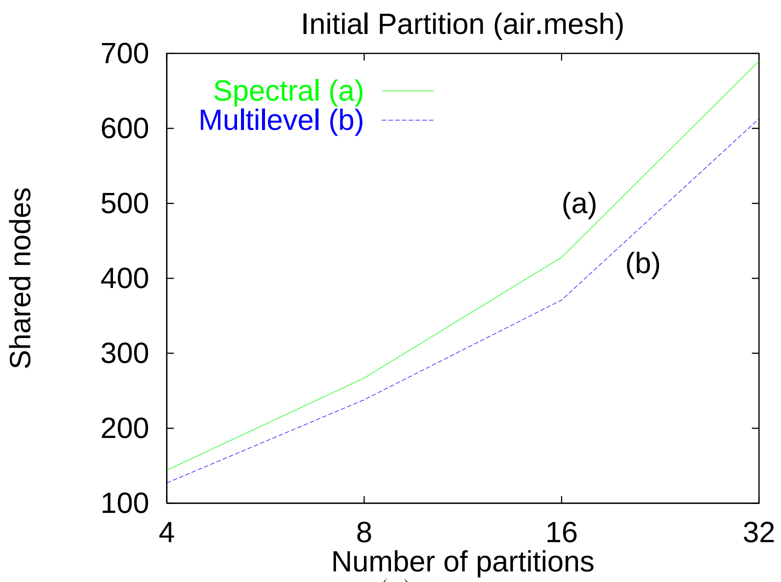


(a)

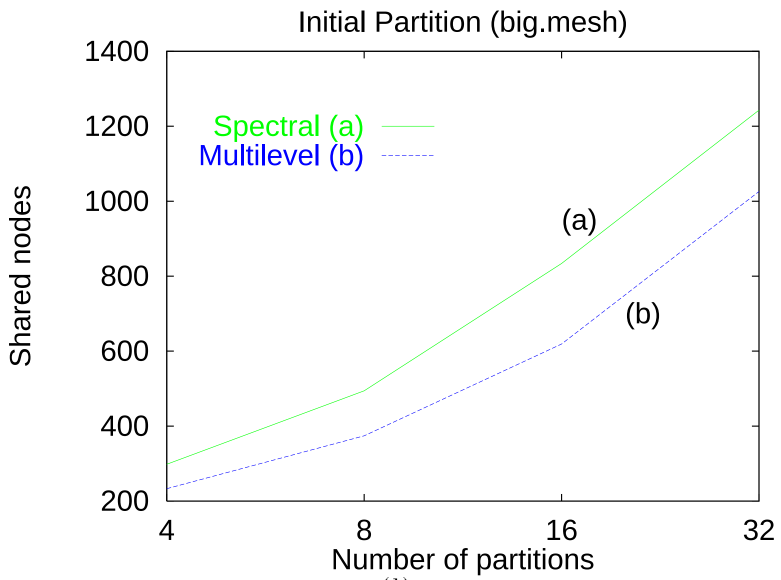


(b)

Figure 36: Partition of the initial mesh using *Chaco*. Time spent to compute 4, 8, 16 and 32 partitions of (a) `air.mesh` and (b) `big.mesh`.



(a)



(b)

Figure 37: Partition of the initial mesh using *Chaco*. Number of shared of nodes after computing the partitions of (a) `air.mesh` and (b) `big.mesh`.

refinement we include the total number of elements, the average number of elements per processor, and the number of elements in the processor that stores the largest number of elements and the one that stores the smallest number. Imbalance is the ratio between the absolute value of the difference between the largest or smallest number of elements (whichever is larger) and the average. Shared nodes is the number of nodes in an internal boundary between processors.

The algorithm spends most of its time in the serial part and the communication cost is very small. This is not surprising because of the way we are selecting the elements for refinement. It is unlikely that by selecting all the elements the refinement is going to propagate to too many processors. In these tests the longest propagation was to a couple of processors. Also note that when we increase the number of processors there is a higher imbalance of the element distribution that reaches a 33 per cent after 6 successive refinements. As it is shown in the figures we are able to obtain superliner speedups. This is due to the fact that when we use a small number of processors we require a lot of memory. This is particularly true when the number of elements assigned to a processor is more than 50,000. In some of these tests we measured 90MB of virtual memory on machines that have around 15MB of physical memory available. This speedup tends to become linear as the number of processors increase and the memory is less of an issue. Figure 38 plots these results and Figure 39 shows the `air.mesh` before and after the refinement of all its elements.

	Number of refinements				
	Initial	**1**	**2**	**3**	**4**
Time (serial)		7.42	35.50	162.63	927.25
Time (comm)		0.13	0.32	0.40	9.80
Time (total)		7.55	35.81	163.03	937.05
Elements (total)	9000	22247	51703	115347	251458
Elements (avg)	2250	5562	12926	28837	62865
Elements (max)	2250	5549	13220	29617	64743
Elements (min)	2250	5522	12793	28461	61949
Imbalance (%)		1.56	2.27	2.70	2.99
Shared nodes	127	194	304	440	676

Table 9: Successive refinements of the `air.mesh` in 4 processors. In each phase all the elements are selected for refinement.

9.5 Migration of the mesh

The migration tests are performed by migrating all the elements in processor P_i to processor P_{i+1}. This is probably one of the most demanding migrations that we can perform. It involves the destruction of all the elements in P_i and the creation of the same elements in P_{i+1}. The timings for this test are shown in Table 13 and plotted in Figure 40. We perform this permutation migration after none, one and two refinements on the `air.mesh` for 4, 8, 16 and 32 processors.

	Number of refinements					
	Initial	**1**	**2**	**3**	**4**	**5**
Time (serial)		2.51	11.68	50.89	223.40	1396.67
Time (comm)		0.10	0.21	0.30	0.67	10.92
Time (total)		2.61	11.89	51.19	224.07	1407.59
Elements (total)	9000	22253	51711	115363	251490	535896
Elements (avg)	1125	2782	6464	14420	31436	66987
Elements (max)	1125	2814	7011	15900	35005	75020
Elements (min)	1125	2718	6205	13708	29708	63009
Imbalance (%)		2.30	8.46	10.26	11.35	11.99
Shared nodes	238	357	558	815	1241	1773

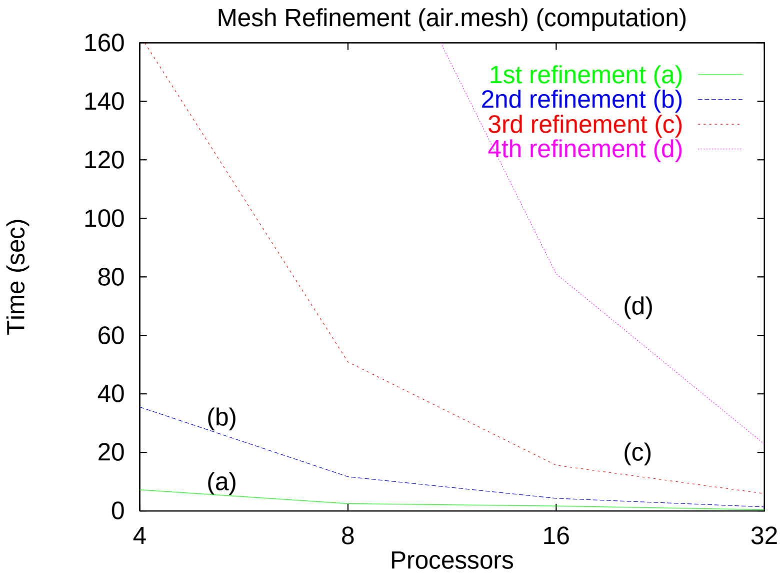
Table 10: Successive refinements of the `air.mesh` in 8 processors. In each phase all the elements are selected for refinement.

	Number of refinements						
	Initial	**1**	**2**	**3**	**4**	**5**	**6**
Time (serial)		1.70	4.30	15.64	81.00	347.69	2535.00
Time (comm)		0.12	0.29	0.31	0.58	1.69	86.84
Time (total)		1.82	4.59	15.95	81.58	349.38	2620.84
Elements (total)	9000	22247	51703	115347	251458	535840	1124496
Elements (avg)	563	1390	3231	7209	15716	33490	70281
Elements (max)	563	1580	3951	9173	20514	44357	93940
Elements (min)	562	13.11	2918	6316	13481	28298	58739
Imbalance (%)		13.70	22.28	27.24	30.53	32.45	33.66
Shared nodes	371	576	911	1292	2044	2795	4357

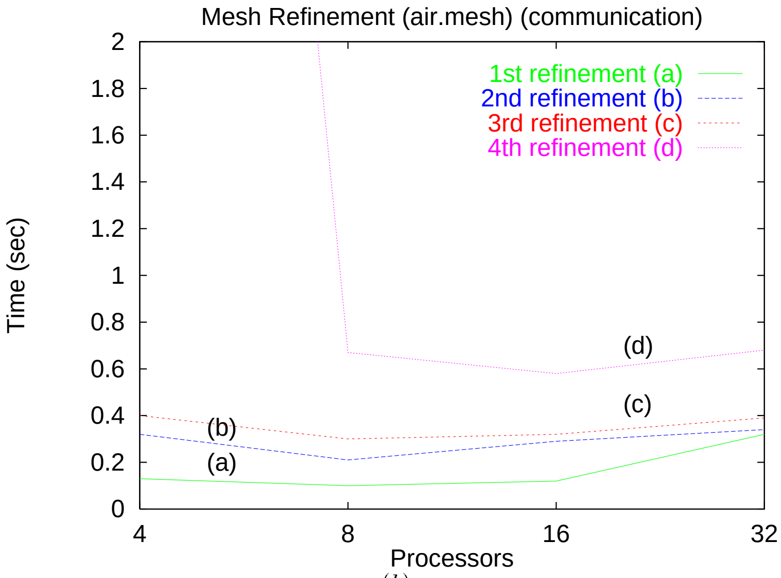
Table 11: Successive refinements of the `air.mesh` in 16 processors. In each phase all the elements are selected for refinement.

	Number of refinements						
	Initial	**1**	**2**	**3**	**4**	**5**	**6**
Time (serial)		0.42	1.38	5.91	22.80	77.66	376.17
Time (comm)		0.32	0.34	0.39	0.68	1.18	4.00
Time (total)		0.73	1.72	6.30	23.48	78.84	380.17
Elements (total)	9000	22251	51713	115363	251490	535902	1124612
Elements (avg)	281	695	1616	3605	7859	16747	35114
Elements (max)	281	788	1967	4567	10214	22080	46764
Elements (min)	282	642	1389	3081	6271	13069	26979
Imbalance (%)		13.38	21.72	26.69	29.97	31.84	33.06
Shared nodes	613	945	1475	2138	3307	4633	7093

Table 12: Successive refinements of the `air.mesh` in 32 processors. In each phase all the elements are selected for refinement.

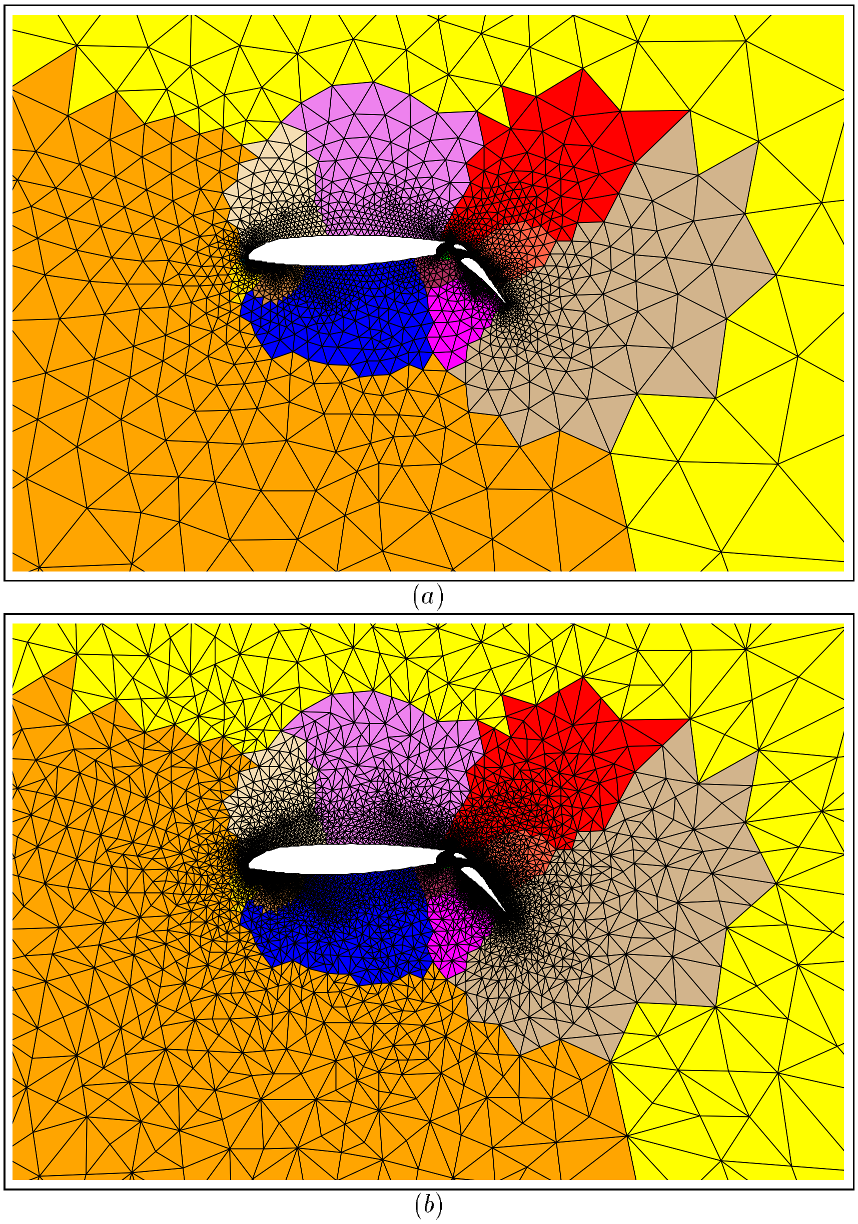


(a)



(b)

Figure 38: Successive refinements of the mesh. In (a) we show time spent in the serial part of the algorithm while in (b) we show the time spent on communication.



(a)

(b)

Figure 39: Refinement of the mesh. The initial `air.mesh` (a) and after refining all its elements (b).

Remember that the refinement of all the elements of the mesh more than doubles the size of the mesh. Also the migration does not include only the elements in the fine mesh but also all the elements in the intermediate meshes. When the mesh is not refined the cost of the algorithm is dominated by the communication. This is why we do not observe any speedup in the column corresponding to the migration of the initial mesh. After one or two refinements we observe linear speedups. The migration of the mesh after two refinements using four processors is a special case. We believe that the low performance of the algorithm at that case is because we are consuming too much memory.

	Migration after successive refinements		
Processors	**Initial mesh**	**1 refinement**	**2 refinements**
4	1.85	37.99	283.60
8	1.76	13.37	79.95
16	2.62	6.44	27.24
32	4.86	5.89	12.26

Table 13: Migration of the mesh. Time in seconds to migrate each element of the `air.mesh` mesh that it is assigned to P_i to P_{i+1} after none, one or two refinements.

Figure 41 shows two stages of this test. In 41 (a) we display a snapshot of the mesh with no refinements before the migration and in (b) we show the same mesh after this permutation migration. Figure 42 (a) shows another migration example. In this case all the elements in each processor are migrated to a random processor. In Figure 42 (b) we migrate the elements according to a target partition.

9.6 Dynamic partitioning of the mesh

Finally we put all the tests together. In this section we present tests that refine the mesh, find a new partition at run time and migrate the mesh according to the new partition. These tests are performed using two different methods. For the first set of tests we partition the mesh using the Parallel Nested Repartitioning algorithm (see Section 7.1.3). In parallel we compute the weight of the mesh M_0^{-1} by collapsing elements that are descendants of a common element of the coarse initial mesh. This phase does not require communication. We then send the dual of the coarse mesh to P_0 to start the serial phase of the algorithm. In the serial phase we find a partition of the coarse mesh using a serial algorithm. In this phase we use *Chaco*'s serial Multilevel Bisection algorithm to partition this reduced graph. We then broadcast the partition to the processors to start the migration phase.

For comparison we run the same tests using the serial Multilevel Bisection algorithm to repartition the mesh. Rather than collapsing the refined elements in parallel as in the PNR algorithm we send to P_0 the whole mesh. In this test we forget that the fine mesh was obtained as the result of the refinement of

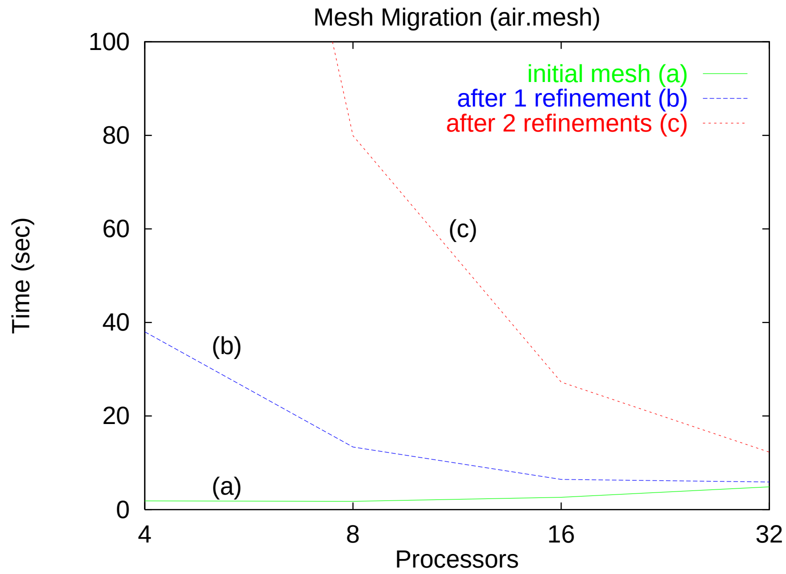
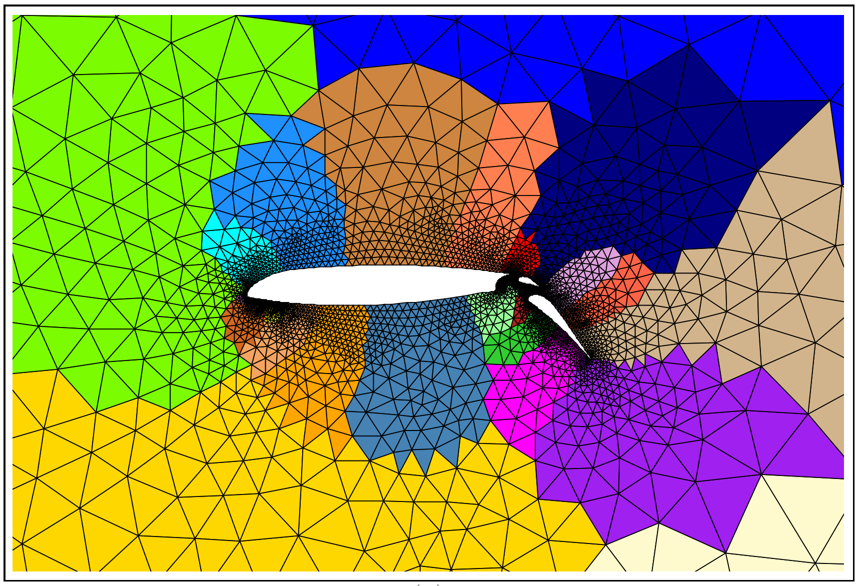


Figure 40: Migration of the mesh. We migrate each element of the `air.mesh` that it is assigned to P_i to P_{i+1} after none, one or two refinements.

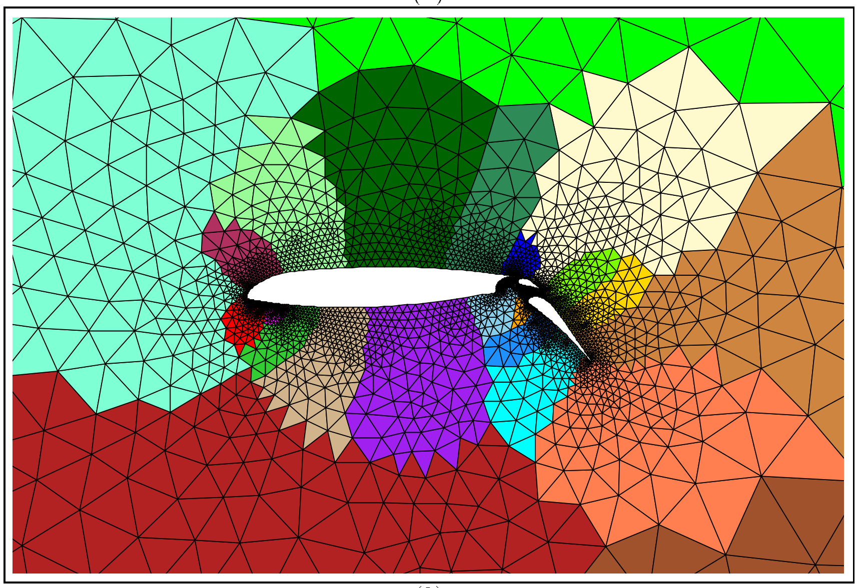
another coarser mesh. For this reason we eliminate all the intermediate elements and we send the fine mesh to P_0. We perform this operation by flattening the hierarchy of elements: we delete all the intermediate refined elements leaving only the elements of the fine mesh. The serial multilevel algorithm running in P_0 collapses the elements by computing a matching of the graph. For a complete description of the method see [21]. Note that if we were implementing this method in a parallel computer we would require to communicate at each level to find a matching of the graph.

The results are shown in Tables 14, 15, 16 and 17. These results are also plotted in Figures 43, 44 and 45. In these pictures and tables we divide the repartitioning in four phases:

- in the first phase every processor calculates the weights of its portion of the coarse mesh and sends it to P_0.
- in the second phase P_0 computes the dual of the graph and spawns a process to run *Chaco*. In the case of the PNR algorithm *Chaco* computes a partition of the reduced mesh M_0^{-1}. In the case of the serial algorithm *Chaco* computes a partition using all the elements of the fine mesh.
- in the third phase P_0 sends the new partition to all the processors.
- finally we migrate the elements according to the new partition.



(a)



(b)

Figure 41: Migration of the mesh. In (a) we show the `air.mesh` distributed between 32 processors. This partition was obtained using a Multilevel Bisection algorithm. In (b) we migrate every element in each processor to the next processor.

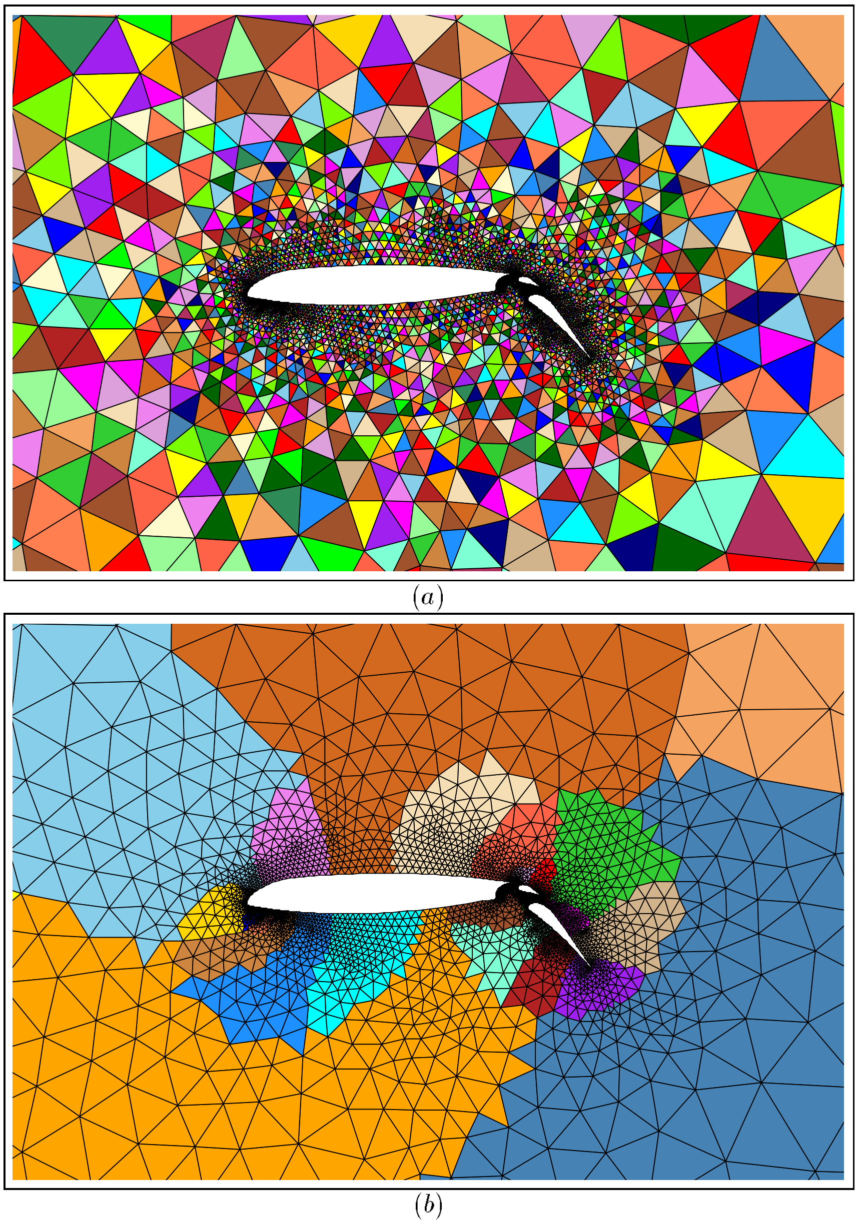


Figure 42: Migration of the mesh. In (a) we migrate the elements to a random processor and in (b) we migrate the mesh according to a Spectral Bisection partition.

First note that if the mesh is not refined both the serial and PNR algorithm are essentially the same and should produce similar results. Only when the mesh is refined we would note a difference between both approaches.

It is not surprising that the partition is significantly faster if we send only the coarse mesh to P_0 as in the PNR algorithm rather than the fine mesh as in the serial algorithm. In the PNR algorithm the cost of computing the weights of the mesh M_0^{-1} and sending it to P_0 does not increase too much as we perform successive refinements of the mesh. In this case $|M_0^{-1}|$ is a constant so the same number of elements are always sent to P_0. This is obviously not true in the serial algorithm. As P_0 needs to partition a much smaller graph during the serial phase of the PNR algorithm the cost of performing this partition is much smaller than if it was made using the serial algorithm. The time to receive the partition from P_0 increases faster in the serial algorithm than in the PNR algorithm as P_0 needs to return longer messages because it partitioned a bigger graph.

It is also not surprising that the migration phase using the serial algorithm performs better than the one using the PNR algorithm because in the serial algorithm we removed all the intermediate elements. In this case the migration corresponding to the PNR partition does not only need to migrate the elements in the fine mesh but also all the refined elements. But this advantage is outweighed by the cost of sending the mesh to P_0 and performing the partition on a bigger mesh.

The really important results are obtained by looking at the last row of Tables 14, 15, 16 and 17 and comparing the quality of the partitions. Our Parallel Nested Repartitioning algorithm produced almost always better partitions than the serial multilevel algorithm and we have shown in Section 9.3 that the serial Multilevel Bisection algorithm produced better partitions than the highly acclaimed Recursive Spectral Bisection algorithm. This proves that the information from the refinement can be effectively used in the mesh partitioning algorithms.

	PNR algorithm			**Serial algorithm**		
	Initial	**1 Ref**	**2 Ref**	**Initial**	**1 Ref**	**2 Ref**
Send mesh to P_0	20.59	32.85	34.85	21.52	57.24	256.91
Partition	18.05	17.81	18.38	17.70	55.47	267.27
Receive partition from P_0	1.17	0.90	1.28	0.94	2.41	39.24
Migration	5.36	25.90	281.53	8.80	22.65	109.53
Total time	46.06	77.89	337.47	49.44	138.75	675.71
Number of shared nodes	119	205	298	119	233	442

Table 14: Repartition of the `air.mesh` in 4 processors after none, one and two refinements using the PNR algorithm and the serial Multilevel Bisection algorithm. Times are in seconds.

	PNR algorithm			Serial algorithm		
	Initial	**1 Ref**	**2 Ref**	**Initial**	**1 Ref**	**2 Ref**
Send mesh to P_0	17.04	16.13	21.16	15.72	49.01	229.20
Partition	22.23	21.62	21.73	22.03	57.19	263.98
Receive partition from P_0	1.83	1.77	1.67	1.67	2.15	6.56
Migration	4.52	16.55	84.08	4.81	8.80	29.66
Total time	47.71	56.23	128.97	44.41	118.00	530.76
Number of shared nodes	221	465	510	221	448	694

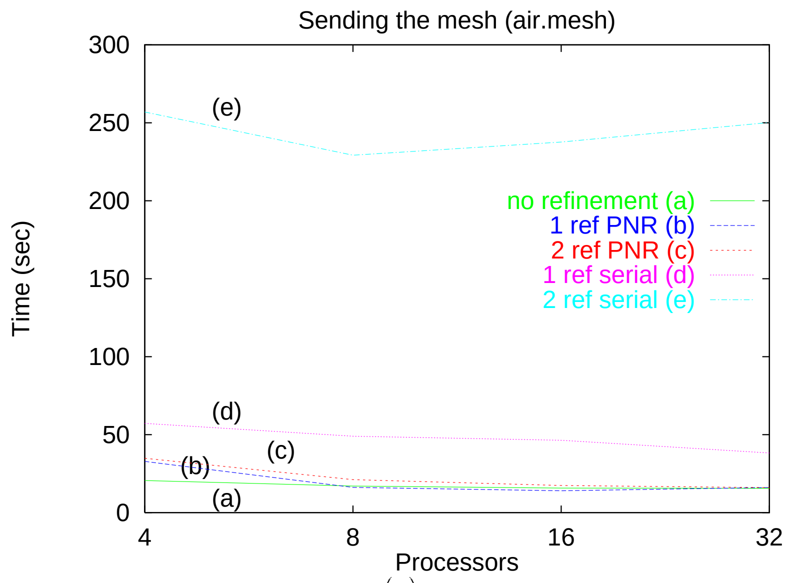
Table 15: Repartition of the `air.mesh` in 8 processors after none, one and two refinements using the PNR algorithm and the serial Multilevel Bisection algorithm. Times are in seconds.

	PNR algorithm			Serial algorithm		
	Initial	**1 Ref**	**2 Ref**	**Initial**	**1 Ref**	**2 Ref**
Send mesh to P_0	15.76	13.99	17.39	14.16	46.39	237.69
Partition	34.51	31.26	33.61	30.96	76.81	304.19
Receive partition from P_0	2.49	2.31	2.95	2.95	3.03	5.50
Migration	7.82	10.36	32.70	6.76	9.52	29.86
Total time	60.69	58.13	86.86	54.97	138.40	591.30
Number of shared nodes	397	597	919	397	797	1225

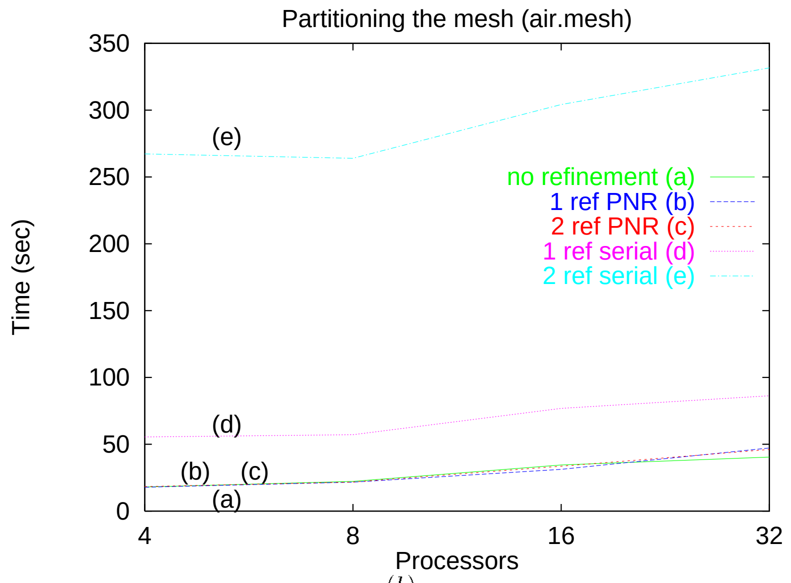
Table 16: Repartition of the `air.mesh` in 16 processors after none, one and two refinements using the PNR algorithm and the serial Multilevel Bisection algorithm. Times are in seconds.

	PNR algorithm			Serial algorithm		
	Initial	**1 Ref**	**2 Ref**	**Initial**	**1 Ref**	**2 Ref**
Send mesh to P_0	15.62	16.09	16.05	14.65	38.25	250.19
Partition	40.44	47.26	46.16	40.72	86.25	331.65
Receive partition from P_0	4.99	5.19	4.42	5.24	5.30	9.61
Migration	23.59	15.74	30.90	21.19	91.90	30.83
Total time	85.26	84.48	97.71	82.01	252.94	624.18
Number of shared nodes	618	955	1525	618	1122	1701

Table 17: Repartition of the `air.mesh` in 32 processors after none, one and two refinements using the PNR algorithm and the serial Multilevel Bisection algorithm. Times are in seconds.

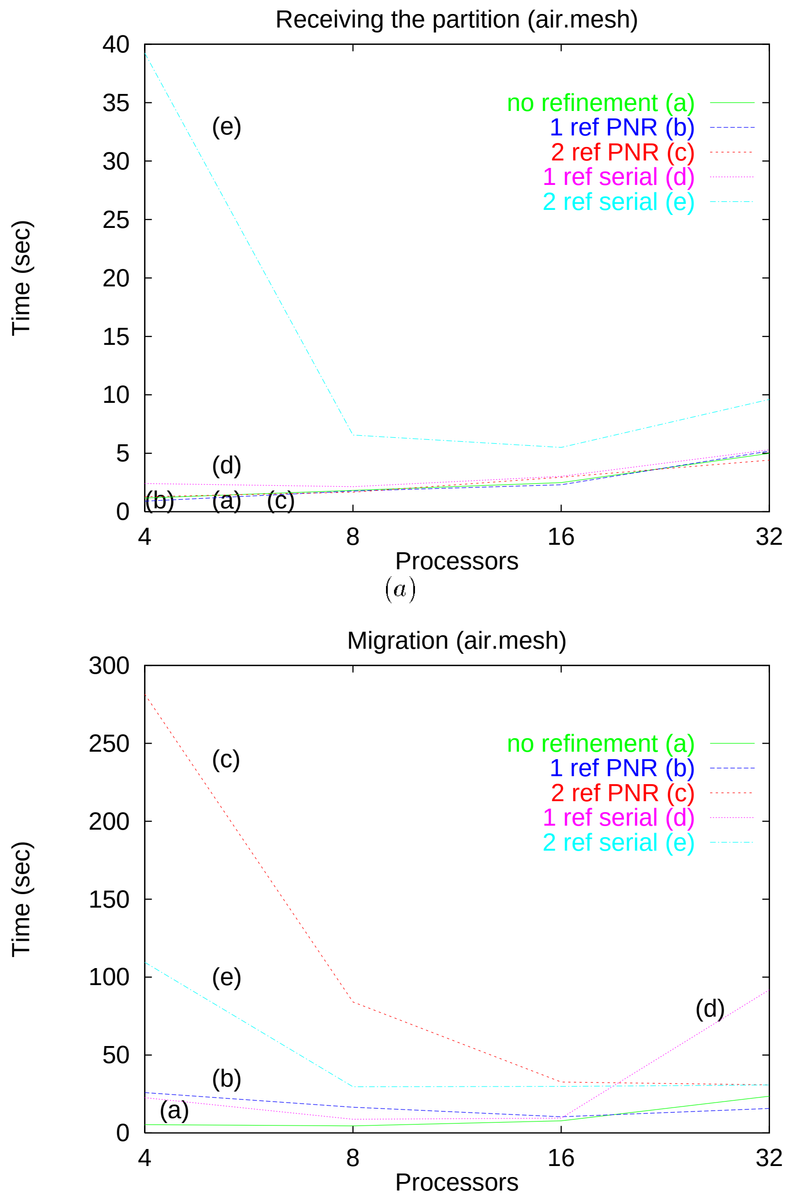


(a)



(b)

Figure 43: Partitioning of the mesh after none, one and two refinements using the PNR algorithm and the serial Multilevel Bisection algorithm. (a) shows the time spent on sending the mesh to one processor. (b) shows the time spent by P_0 to partition the graph.



(a)

(d)

Figure 44: Partitioning of the mesh after none, one and two refinements using the PNR algorithm and the serial Multilevel Bisection algorithm. (a) is the time spent on communicating back the results of the partition from P_0 to the processors and (b) shows the time spent on the migration of the mesh.

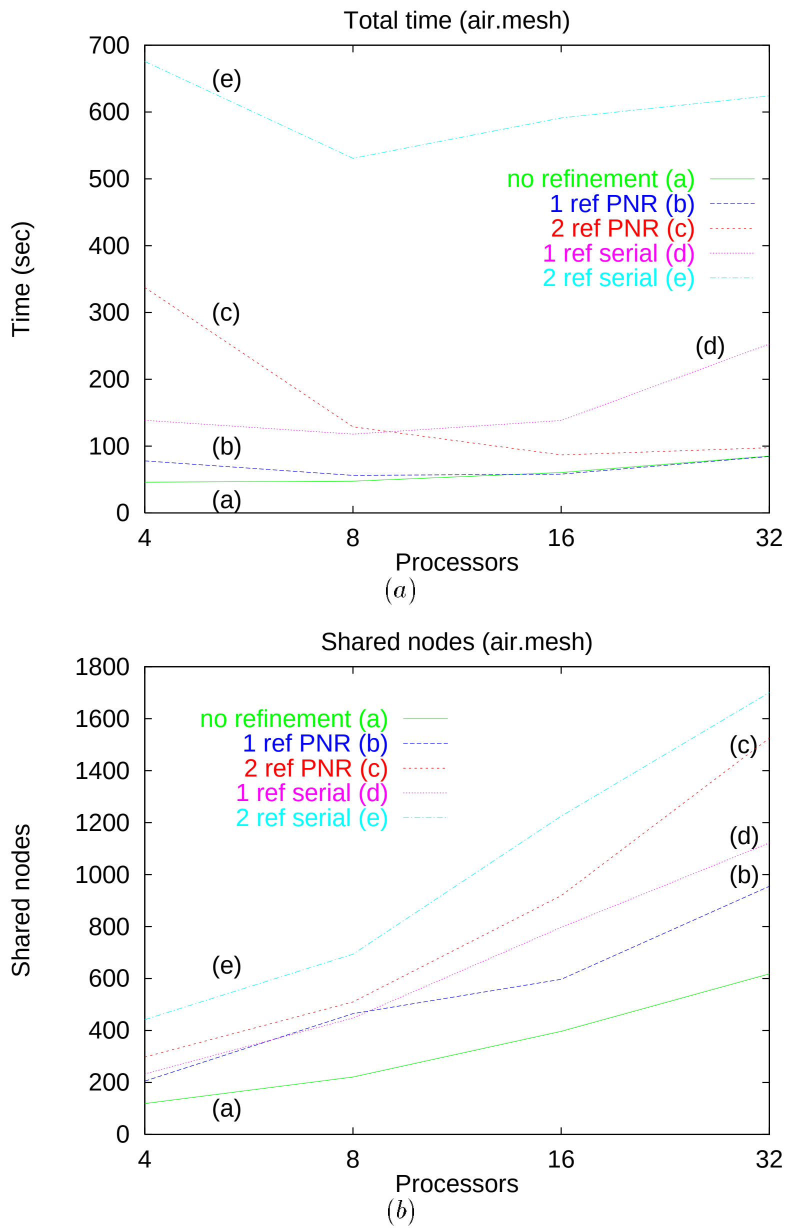


Figure 45: Partitioning of the mesh after none, one and two refinements using the PNR algorithm and the serial Multilevel Bisection algorithm. (a) is the total time. The number of shared nodes is shown in (b).

10 Related projects

This project follows the spirit of the Distributed Irregular Mesh Environment (DIME) [31], [43] by Roy Williams at the California Institute of Technology. DIME allowed the refinement of triangles but was not able to coarsen them. Also it is not clear how its parallel refinement and load migration work.

The Scalable Unstructured Mesh Computation (SUMAA3d) at the Argonne National Laboratories is another related project. The refinement algorithm [3] avoids the creation of duplicate nodes in the boundaries of processors by refining the elements in independent sets so that adjacent elements in different processors are never refined at the same time. These independent sets are computed using Monte Carlo methods. Also the partitioning algorithm is based on the Orthogonal Recursive Bisection method. In general is possible to obtain much better partition using multilevel methods.

In [33], [42] an adaptive environment for unstructured problems is presented. In this project the migration is only done to adjacent processors using iterative local migration techniques. The load balancing algorithm is done by comparing the work between adjacent processors. This means that we might require several iterations to rebalance the work.The refinement is performed using quadtree structures.

11 Conclusion and future work

In this thesis we present contributions in the areas of mesh refinement, mesh repartitioning and mesh migration.

We have developed parallel refinement algorithms for unstructured meshes and used the refinement history to develop a Parallel Nested Repartitioning algorithm superior to the algorithms in [19] when applied to the partition of adapted meshes. Although we explained the theoretical problems of doing the refinement in parallel our tests showed very little overhead due to communication.

We used the Parallel Nested Repartitioning algorithm to compute partitions on the fly. We showed that we can obtain high quality partitions at a reasonable cost. We also showed that we can use these partitions to rebalance the work by migrating elements and nodes between the processors.

The implementation of the project was highly simplified by using C++. The use of an object oriented language allowed us to design irregular data structures efficiently. It also reduced the number of bugs as we encapsulated the most dangerous components (like the remote pointers) into objects.

There are two major possible improvements to this project. The first one is to port it to the IBM SP-2. The second one is to find a real physical problem to drive the computation. Although we have worked only with triangles we designed the data structures to allow different types of elements both in 2D and in 3D.

References

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

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

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

[4] Dimitri P. Bertsekas and John N. Tsisiklis: *Parallel and Distributed Computation: Numerical Methods*, 570-579, Prentice Hall, New Jersey, 1989.

[5] Ivo Babuska, Jagdish Chandra and Joseph E. Flaherty (eds.): *Adaptive computational methods for partial differential equations*, SIAM, Philadelphia, 1983.

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

[7] L. Ferragut, R. Montenegro and L. Plaza: *Efficient refinement-derefinement algorithm of nested meshes to solve evolution problems*, Communications in Numerical Methods in Engineering, Vol. 10, 403-412, 1994.

[8] Kenneth G. Powell, Philip L. Roe and James Quirk: *Adaptive-mesh algorithms for computational fluid dynamics*, Algorithmic Trends in Computational Fluid Dynamics, Springer-Verlag, 1991.

[9] J. E. Savage and M. Wloka: *Parallelism in Graph Partitioning*. Journal of Parallel and Distributed Computing, 13, 257-272, 1991.

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

[11] A. Bowyer: *Computing Dirichlet tessalations*, Comp. J., 24, 162, 1981.

[12] J. E. Akin: *Finite Elements for Analysis and Design*, Academic Press, 1994.

[13] 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 solution of PDEs*, Proc. 1992 DAGS Symposium, 1992.

[14] John J. Barton and Lee R. Nackman: *Scientific and Engineering C++: An introduction with advanced techniques and examples*, Addison Wesley, 1994.

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

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

[17] S. T. Barnard and H. D. Simon: *A parallel implementation of multilevel recursive spectral bisection for application to adaptive unstructured meshes*, Proceedings of the 7th SIAM conference on Parallel Processing for Scientific Computing, 627-632, 1995.

[18] G. Karypis and V. Kumar: *A fast and high quality multilevel scheme for partitioning irregular graphs*, Tech. Rep. CORR 95-035, University of Minnesota, Dept. of Computer Science, 1995.

[19] G. Karypis and V. Kumar: *Parallel Multilevel Graph Partitioning*, Tech. Rep. CORR 95-036, University of Minnesota, Dept. of Computer Science, 1995.

[20] B. Hendrickson and R. Leland: *A multilevel algorithm for partitioning graphs*, Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[21] B. Hendrickson and R. Leland: *The Chaco user's guide*, Technical Report SAND93-2339, Sandia National Laboratories, 1993.

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

[23] Message Passing Interface Forum: *MPI: A Message Passing Interface Standard*, 1994.

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

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

[26] W. Gropp and E. Lusk: *User's Guide for MPICH: A portable Implementation of MPI*, Argonne National Lab and Missisipi State University.

[27] R. Butler and E. Lusk: *User's guide to the p4 Parallel Programming System*, Technical Report ANL-92/17, Argonne National Laboratory, 1992.

[28] B. Welch: *Practical Programming in Tcl and Tk*, Prentice Hall, 1995.

[29] J. Ousterhout: *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[30] Sun Microsystems: *Tools.h++ Class Library: Introduction and Reference Manual*, SunPro, 1993.

[31] R. D. Williams: *DIME: A User's Manual*, Caltech Concurrent Computation Report C3P 861, 1990.

[32] B. Kernighan and S. Lin: *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, 29, 291-307, 1970.

[33] K. Devine, J. Flaherty, R. Loy and S. Wheat: *Parallel partitioning strategies for the adaptive solution of conservation laws*, Modeling, Mesh Generation and Adaptive Numerical Methods for Partial Differential Equations, I. Babuska, W. Henshaw, J Oliger, J. Flaherty, J. Hopcroft and T. Tezduyar (eds.), Springer-Verlag, 1993.

[34] R. Lohner: *Mesh Adaptation in Fluid Mechanics*, Engineering Fracture Mechanics, Pergammon, Vol 50, Num 5, 819-847, 1995.

[35] L. Freitag, M. Jones and P. Plassmann: *An Efficient Parallel Algorithm for Mesh Smoothing*, Proceedings of the 4th International Meshing Roundtable, Sandia National Laboratories, 47-58, 1995.

[36] N. Hitschfeld: *Algorithms and Data Structures for Handling a Fully Flexible Refinement Approach in Mesh Generation*, Proceedings of the 4th International Meshing Roundtable, Sandia National Laboratories, 265-276, 1995.

[37] M. Price, C. Stops and G. Butlin: *A Medial Object Toolkit for Meshing and Other Applications*, Proceedings of the 4th International Meshing Roundtable, Sandia National Laboratories, 219-229, 1995.

[38] D. Vanderstraeten and R. Keunings: *Optimized Partitioning of Unstructured Finite Element Meshes*, International Journal for Numerical Methods in Engineering, John Wiley and Sons, Ltd., Vol 38, 433-450, 1995 .

[39] P. L. George: *Automatic mesh generation : application to finite element method*, Wiley and Sons, Ltd., 1991.

[40] P. Knupp and S. Steinberg: *Fundamentals of grid generation*, CRC Press, 1993.

[41] L. A. Freitag, M. T. Jones and P. E. Plassmann: *New advances in the modeling of high-temperature superconductors*, 1994 International Simulation Conference, Grand Challenges in Computer Simulation, 208-213, La Jolla, California, 1994.

[42] 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.

[43] G. C. Fox, R. D. Williams and P. C. Messina: *Parallel Computing Works!*, Morgan Kaufmann Publishers, Inc. 1994.