The Dynamic Adaptation of Parallel Mesh-Based Computation

José G. Castaños* John E. Savage *

Abstract

We present an overview of 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. 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 paper 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]. We propose a new Parallel Nested Repartitioning algorithm that has its roots in the multilevel bisection algorithm of Barnard et al [3]. It produces high quality partitions at a low cost, a very important requirement for recomputing partitions at runtime.

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

*Brown University, Dept. of Computer Science

to another in order to rebalance the work. The mesh is represented as a set of C++ objects. To permit references to mesh elements and nodes on a foreign processor, local proxies for elements and nodes are used.

To evaluate these ideas we designed and implemented a system in C++. This program runs on a network of workstations and uses MPI [4] 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 has allowed us to reduce the complexity of the implementation without significantly affecting performance.

2 Multilevel mesh adaptation

To support dynamic adaptation of meshes we designed a data structure based on a multilevel finite-element mesh. 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 vertices. Using defined *adaptation criteria* we select 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.

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 at time $t > 0$ each element that it is not refined belongs to the *fine mesh* M_t.

The selection of elements for refinement (or coarsening) is made by examining the values of an adaptation criterion [5]. 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. The Rivara bisection refinement algorithm applies to 2-dimensional triangular meshes. Its simplest form bisects the longest edge of a triangle to form two new triangles.

3 The challenge of exploiting parallelism

We now describe problems introduced by parallelism that need to be solved in order to perform the dynamic adaptation of parallel mesh-based computation.

Refinement algorithms typically use local information to perform refinement. Since the refinement of an element E_a may create a new node V_n in an internal boundary between two processors, synchronization between the processors is necessary.

Detection of termination of a refinement phase must be done globaly because refinement can propagate to adjacent processors. The adaptation of a 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. Finally we must keep a consistent mesh while migrating elements and nodes between processors.

In the following sections we sketch our solution to these problems. We begin with definitions and a strategy for storing meshes in a distributed memory parallel computer.

4 Implementing a parallel mesh using remote references

We assume that meshes are partitioned between processors using an *element-partitioning* method. We denote with $\Pi = \{\Pi_1, \Pi_2, \ldots, \Pi_p\}$ a partition of the elements between

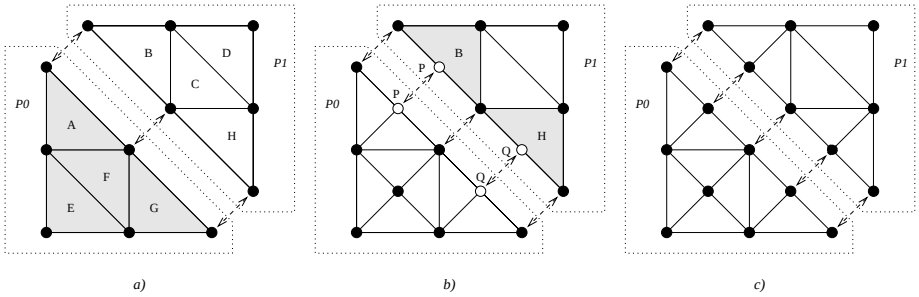


Figure 1: Propagation of refinement to adjacent processors. In (a) the elements E_a, E_e, E_f and E_g are selected for refinement. The refinement of these elements creates two nodes, V_p and V_q, on the boundary between P_0 and P_1. P_1 creates its local copies V_p^1 and V_q^1 and selects the nonconforming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

processors where $\bigcup\ \Pi_i = E$ and $\Pi_i\ \cap\ \Pi_j = \varnothing,\ \forall\ i \neq j.\ E_a \in \Pi_i$ consists of elements assigned to processor P_i.

A remote reference is a pair (P_i, V_n^i) where P_i is a processor and V_n^i is the copy of node V_n in processor P_i. Let $Ref(V_n^j) = \{(P_i, V_n^i) : V_n^i \text{ is a copy of } V_n \text{ in } P_i, i \neq j\}$. This relation is symmetric, i.e. if $(P_j, V_n^j) \in Ref(V_n^i)$ then $(P_i, V_n^i) \in Ref(V_n^j)$. If V_n is a node internal to a processor, then $Ref(V_n) = \varnothing$. A remote reference is a pointer 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*.

The design of remote references is highly influenced by the element partitioning method. As we shall show, 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 copies of itself 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.

5 Parallel mesh adaptation

Using the above data structures we now introduce an algorithm to adapt 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. 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 processors as suggested in Figure 1.

5.1 Refinement collision

A parallel refinement algorithm can run into two synchronization problems [6]. 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 the nodes distinct 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. It may also be that a processor P_i believes element E_b in processor P_j is nonconforming although it might have already been refined there. To avoid this problem processor P_j needs to evaluate and update the propagation requests it receives before executing them.

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 at P_j from P_i requesting that a copy V_n^j be made in processor P_j that causes the refinement of element E_b can be compared against the status of E_b. If E_b was already refined in the local phase (but processor P_i did not know about this), then E_b might not need to be refined again. If node V_n was already created in the local phase of P_j then a reference is added pointing to its copy V_n^i in P_i. If the refinement of E_b did not cause or was not caused by the creation of node V_n 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 using the shared node V_n^j.

5.2 Termination detection

Termination is reached when no element is marked in any processor for refinement. This 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 of the termination algorithm we assume that the refinement does not propagate cyclically from P_i to P_j and then from P_j back to P_i.

At $t = 0$ P_C sends a message to each P_i indicating that the refinement phase has started. P_C can explicitly select the elements for refinement or it can instruct the processors to select elements based on an adaptation criterion. P_i then executes the serial refinement algorithm on these marked elements, possibly sending *refine* messages to neighboring processors when a node V_n^i is created in an internal boundary between them.

The termination detection procedure is based on message acknowledgments. In particular *refine* messages received by P_j from P_i are acknowledged by P_j by sending to P_i an ACK message. If a *refine* message from P_i to P_j causes the refinement to propagate to another processor P_k then P_j sends an ACK message to P_i only after it has received an ACK from P_k. The ACK messages return the references to the newly created vertices so that if V_n^i is a vertex in P_i over a shared edge that caused a propagation to P_j and V_n^j is its copy in P_j, a reference to V_n^j is added at V_n^i and vice versa.

6 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 6.2 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.

6.1 The mesh repartitioning problem

While the PNR algorithm is based on the serial multilevel algorithms presented in [7], [8] and [9] 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 graph. The PNR method differs from

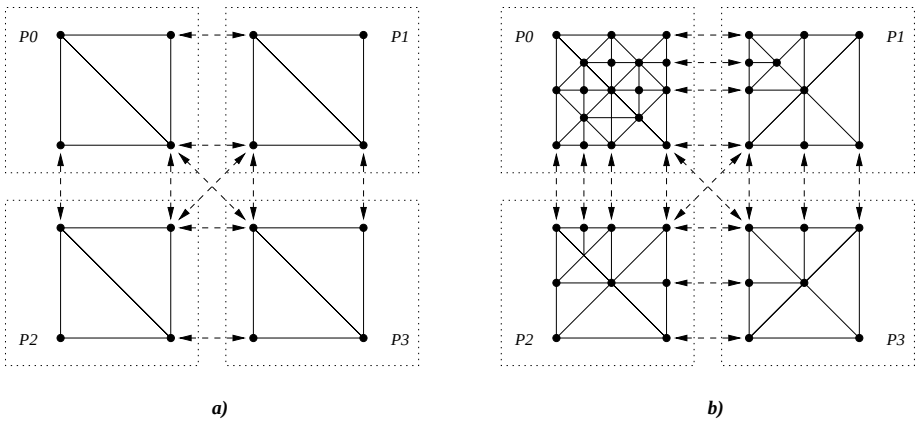


Figure 2: The Parallel Nested Repartitioning algorithm. (a) shows the initial mesh M_0 and (b) shows the mesh M_t at the beginning of the partitioning phase.

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.

6.1.1 The Parallel Nested Repartitioning (PNR) algorithm

The PNR algorithm uses the fact that the fine mesh M_t at time t was obtained as a sequence of refinements of a coarse initial mesh M_0. M_t includes all the elements that have no children at time t. We assume that $|M_t| \gg |M_0|$.

Figure 2 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 much 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.

PNR allows for a very natural parallel implementation. We compute locally in parallel a weighted graph M_0^{-1} that is the dual of the coarse mesh M_0. Each vertex in this graph corresponds to an element of M_0. The weight of a vertex of M_0^{-1} is number of descendants of the element in M_t and the weight of an edge is the number of shared nodes between their descendant leaves. We then send the graph M_0^{-1} to P_C which partitions M_0^{-1}. All the other processors wait until P_C sends back a message informing them of which elements to migrate. PNR has a *parallel* and a *serial* phase:

- M_0^{-1} is constructed in parallel with no processor communication. Weights are computed using a simple recursive algorithm. Once P_i obtains its portion of M_0^{-1} it sends it to P_C for the serial part of the algorithm.

- Once P_C receives a message from each P_i it partitions M_0^{-1} using a serial partitioning algorithm. Since $|M_0^{-1}|$ is assumed to be relatively small we use algorithms that would be considered too expensive to apply to M_t.

- Finally P_C sends a message to each processor P_i informing it of which elements to migrate. P_i then executes the migration algorithm described bellow.

6.2 Using remote references for work migration

We now present an algorithm that migrates elements and nodes between processors. To adjust the mesh according to a new partition we need to move one or more elements E_a from P_i to P_j. Let's assume that the vertices of E_a are $Adj(E_a) = \{V_{n_1}, \ldots, V_{n_q}\}$.

Our algorithm takes into account whether P_j has a local copy of these nodes or not.

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

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

- If processor P_i sends elements E_a and E_b to P_j and there is a node $V_n \in Adj(E_a) \cap Adj(E_b)$ (V_n 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 E_a and E_b should refer to it.

- If P_i and P_k send the adjacent elements E_a and E_b respectively to processor P_j, and there is a node $V_n \in Adj(E_a) \cap Adj(E_b)$, $(P_k, V_n^k) \in Ref(V_n^i)$ and $(P_i, V_n^i) \in Ref(V_n^k)$ and P_j should detect that V_n^i and V_n^k are copies of the same node.

- Finally if P_i sends an element E_a to P_j and P_k sends element E_b to P_l where E_a and E_b are adjacent elements sharing a common node V_n, we must insure that V_n^k and V_n^l refer to each other.

7 Experimental results

To evaluate the quality and performance of our system we performed a series of tests on a network of SUN SparcStation10 workstations. The tests covered 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. PNR computed very high quality partitions in a very reasonable time. In fact we were able to obtain better partitions than any other multilevel algorithm.

7.1 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 were selected for refinement. The number of elements grew exponentially with the level of refinement. By doing a series of successive refinements we were able to create meshes containing more than 2,000,000 elements.

The results are shown in Figure 3. 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. The refinement algorithm spends most of its time in the serial part and the communication cost is very small.

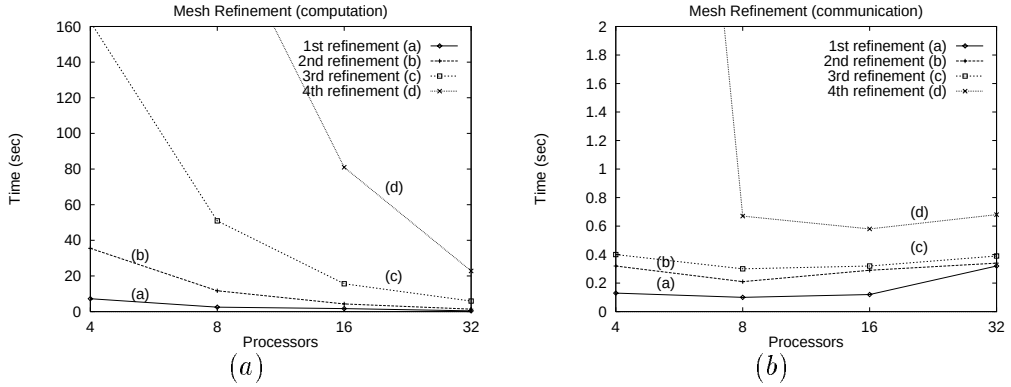


Figure 3: Successive refinements of the mesh. In (a) we show time spent in the serial part of the algorithm while in (b) we show the time spent on communication.

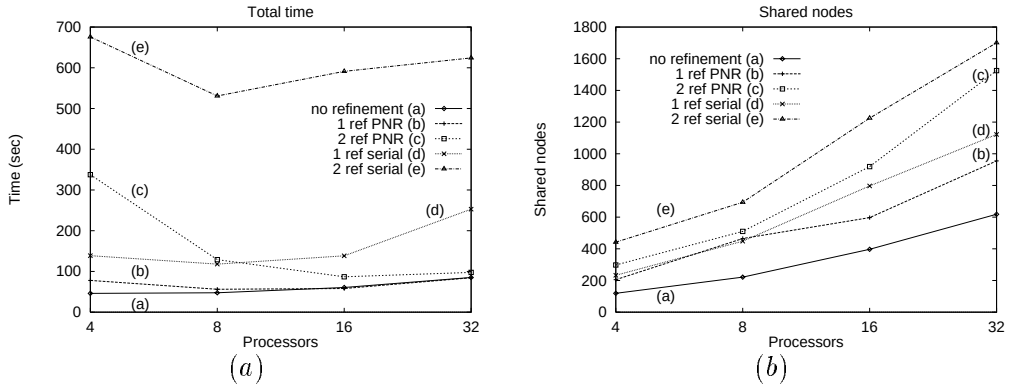
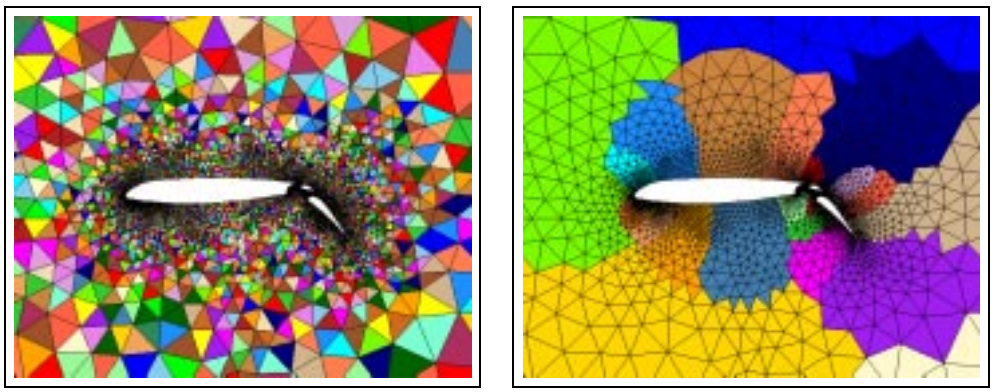


Figure 4: Partitioning of the mesh after none, one and two refinements using the PNR algorithm and the *Chaco* serial Multilevel Bisection algorithm. (a) is the total time. The number of shared nodes is shown in (b).

7.2 Dynamic partitioning of the mesh

We compared the partitioning of refined meshes using the PNR algorithm to that provided by serial multilevel algorithm in *Chaco* [10]. These results are shown in figure 4. The PNR not only performed faster than the serial multilevel algorithm as we increased the refinement of the mesh but it also produced partitions of better quality. This shows that the information from the refinement history can be effectively used to partition the mesh.

Finally, Figure 5 shows an example where the partition of the mesh is computed at runtime and the elements and nodes are migrated according to this new partition.



(a) (b)

Figure 5: Migration of the mesh. In (a) the elements are assignment to the processors at random and in (b) a new partition is computed at runtime.

References

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

[2] M. C. 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] 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.

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

[5] 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.

[6] 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.

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

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

[9] 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.

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