# Parallel Tetrahedral Mesh Adaptation with Dynamic Load Balancing [1]

Leonid Oliker

*NERSC, MS 50B-2239, Lawrence Berkeley National Laboratory, Berkeley, CA 94720. E-mail: loliker@lbl.gov*

Rupak Biswas

*MRJ Technology Solutions, MS T27A-1, NASA Ames Research Center, Moffett Field, CA 94035. E-mail: rbiswas@nas.nasa.gov*

Harold N. Gabow

*Department of Computer Science, University of Colorado, Boulder, CO 80309. E-mail: hal@piper.cs.colorado.edu*

**Abstract**

The ability to dynamically adapt an unstructured grid is a powerful tool for efficiently solving computational problems with evolving physical features. In this paper, we report on our experience parallelizing an edge-based adaptation scheme, called 3D_TAG, using message passing. Results show excellent speedup when a realistic helicopter rotor mesh is randomly refined. However, performance deteriorates when the mesh is refined using a solution-based error indicator since mesh adaptation for practical problems occurs in a localized region, creating a severe load imbalance. To address this problem, we have developed PLUM, a global dynamic load balancing framework for adaptive numerical computations. Even though PLUM primarily balances processor workloads for the solution phase, it reduces the load imbalance problem within mesh adaptation by repartitioning the mesh after targeting edges for refinement but before the actual subdivision. This dramatically improves the performance of parallel 3D_TAG since refinement occurs in a more load balanced fashion. We also present optimal and heuristic algorithms that, when applied to the default mapping of a parallel repartitioner, significantly reduce the data redistribution overhead. Finally, portability is examined by comparing performance on three state-of-the-art parallel machines.

[1] Work supported by NASA under Contract Numbers NAS 2-96027 with Universities Space Research Association and NAS 2-14303 with MRJ Technology Solutions.

## 1 Introduction

Unstructured grids[2] for solving computational problems have two major advantages over structured grids. First, unstructured meshes enable efficient grid generation around highly complex geometries. Second, appropriate unstructured-grid data structures facilitate the rapid insertion and deletion of points to allow the mesh to locally adapt to the solution.

Two solution-adaptive strategies are commonly used with unstructured-grid methods. Regeneration schemes generate a new grid with a higher or lower concentration of points in different regions depending on an error indicator. A major disadvantage of such schemes is that they are computationally expensive. This is a serious drawback for unsteady problems where the mesh must be frequently adapted. However, resulting grids are usually well-formed with smooth transitions between regions of coarse and fine mesh spacing.

Local mesh adaptation, on the other hand, involves adding points to the existing grid in regions where the error indicator is high, and removing points from regions where the indicator is low. The advantage of such strategies is that relatively few mesh points need to be inserted or deleted at each refinement/coarsening step for unsteady problems. However, complicated logic and data structures are required to keep track of the points that are added and removed.

For problems that evolve with time, local mesh adaptation procedures have proved to be robust, reliable, and efficient. By redistributing the available mesh points to capture physical phenomena of interest, such procedures make standard computational methods more cost effective. Highly localized regions of mesh refinement are required in order to accurately capture shock waves, contact discontinuities, vortices, and shear layers. This provides scientists the opportunity to obtain solutions on adapted meshes that are comparable to those obtained on globally-refined grids but at a much lower cost. Even though adaptive mesh algorithms are commonly used for problems in fluid flow and structural mechanics, they are also of significant interest in several other areas like computer vision and graphics.

Advances in adaptive software and methodology notwithstanding, parallel computational strategies will be an essential ingredient in solving complex real-life problems. However, parallel computers are usually easier to program with regular data structures; so the development of efficient parallel adaptive algorithms for unstructured grids (that use complex data structures and indirect addressing) poses a serious challenge. Their parallel performance for supercomputing applications not only depends on the design strategies, but also on the

[2] The terms *grid* and *mesh* are used synonymously throughout this paper.

choice of efficient data structures which must be amenable to simple manipulation without significant memory contention (for shared-memory architectures) or communication overhead (for message-passing architectures). Nonetheless, it is generally believed that adaptive unstructured-grid techniques will constitute a significant fraction of future high-performance computing.

A significant amount of research has been done to design sequential algorithms to effectively use unstructured meshes for fluid flow applications, e.g., the solution of the Euler equations. Unfortunately, many of these techniques cannot take advantage of the power of parallel computing due to the difficulties of porting these codes onto distributed-memory architectures. Recently, several adaptive schemes have been successfully developed in a parallel environment. Most of these codes are based on two-dimensional finite elements [1,2,5,9,17,18,25], and some progress has been made towards three-dimensional unstructured-mesh simulations [4,21,27,28]. Various dynamic load balancing methods for unstructured-grid applications have also been reported to date [10–12,16,22,32–34]; however, most of them lack a global view of loads across all processors.

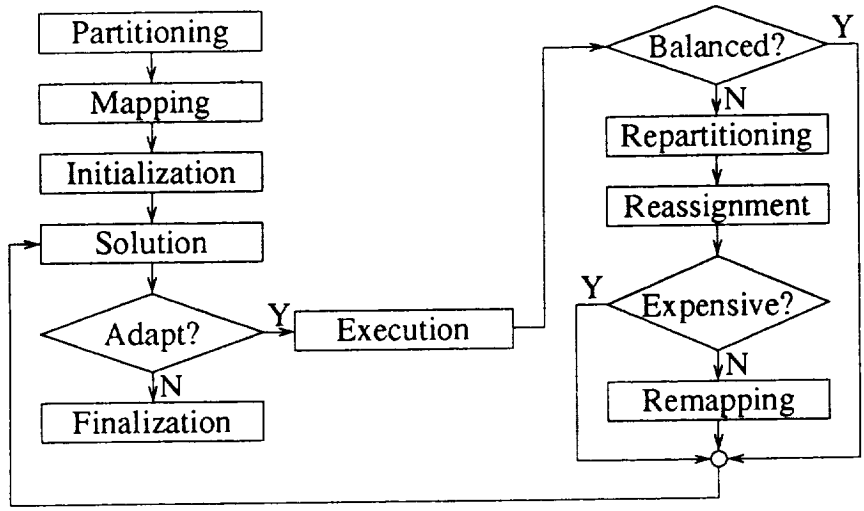



Fig. 1. Overview of our global dynamic load balancing framework for adaptive numerical computations.

Figure 1 depicts our global dynamic load balancing framework for adaptive computations. It essentially consists of a numerical solver and our mesh adaptor, with a partitioner and a remapper that load balance and redistribute the computational mesh when necessary. The mesh is first partitioned and mapped among the available processors. The initialization phase distributes the global data among the processors and generates a database for all shared objects[3]. The numerical solver then runs for several iterations, updating solution variables that are typically stored at the vertices of the mesh. When an acceptable

[3] The term *object* is used generically to denote a vertex, edge, tetrahedron, or face in the mesh.

solution is obtained, local mesh adaptation is performed to generate a new computational mesh, if so desired. A quick evaluation step determines if the new mesh is sufficiently unbalanced to warrant a repartitioning. If the current partitioning indicates that it is adequately load balanced, control is passed back to the solver. Otherwise, a mesh repartitioning procedure is invoked to divide the new grid into subgrids. The new partitions are then reassigned to the processors in a way that minimizes the cost of data movement. If the cost of remapping the data is less than the computational gain that would be achieved with balanced partitions, all necessary data is appropriately redistributed. Otherwise, the new partitioning is discarded and the calculation continues on the old partitions. The finalization step combines the local grids on each processor into a single global mesh. This is usually required for some post-processing tasks, such as visualization, or to save a snapshot of the grid on secondary storage for future restart runs.

Notice from the framework in Fig. 1 that the computational load is balanced and the runtime communication reduced only for the solver but not for the mesh adaptor. This is important since solvers are usually several times more expensive. However, parallel performance for the mesh adaptation procedure can be significantly improved if the mesh is repartitioned and remapped in a load-balanced fashion after edges are targeted for refinement and coarsening but before performing the actual adaptation. This strategy also reduces the redistribution cost significantly since a smaller volume of data is moved.

The numerical solver is usually application-dependent, and is beyond the scope of this paper. Here, we focus on some of the tools that enable numerical simulations to be accomplished rapidly and efficiently. Parallel mesh adaptation and dynamic load balancing are two such critical tools.

## 2 Tetrahedral Mesh Adaptation

We first give a brief description of the tetrahedral mesh adaptation scheme [7] that is used in this work to better explain the modifications that were made for the distributed-memory implementation. The 5,000-line C code, called 3D_TAG, has its data structures based on edges that connect the vertices of a tetrahedral mesh. This means that the elements[4] and boundary faces are defined by their edges rather than by their vertices. These edge-based data structures make the mesh adaptation procedure capable of efficiently performing anisotropic refinement and coarsening. A successful data structure must contain the right amount of information to rapidly reconstruct the mesh connectivity when vertices are added or deleted while having reasonable memory

[4] The terms *element* and *tetrahedron* are used synonymously throughout this paper.

requirements.

Recently, the 3D_TAG code has been modified to refine and coarsen hexahedral meshes [8]. The data structures and serial implementation for the hexahedral scheme are similar to those for the tetrahedral code. Their parallel implementations should also be similar; however, this paper focuses solely on tetrahedral mesh adaptation.

### *2.1 The Algorithm*

At each mesh adaptation step, individual edges are marked for coarsening, refinement, or no change, based on an error indicator calculated from the solution. Edges whose error values exceed a user-specified upper threshold are targeted for subdivision. Similarly, edges whose error values lie below another user-specified lower threshold are targeted for removal. Only three subdivision types are allowed for each tetrahedral element and these are shown in Fig. 2. The 1:8 isotropic subdivision is implemented by adding a new vertex at the mid-point of each of the six edges. The 1:4 and 1:2 subdivisions can result either because the edges of a parent tetrahedron are targeted anisotropically or because they are required to form a valid connectivity for the new mesh. When an edge is bisected, the solution quantities are linearly interpolated at the mid-point from its two end-points.

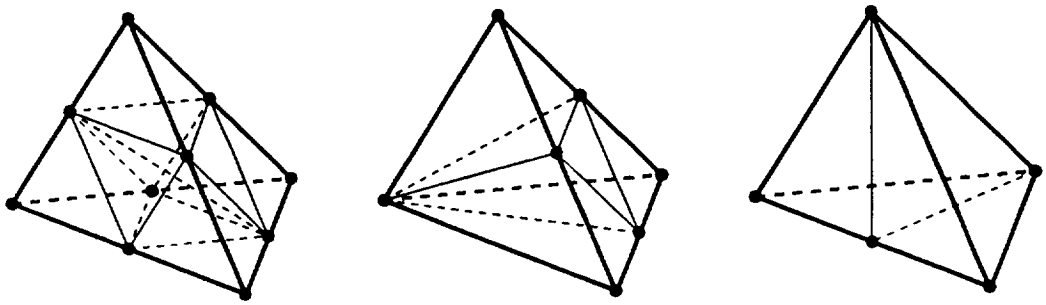
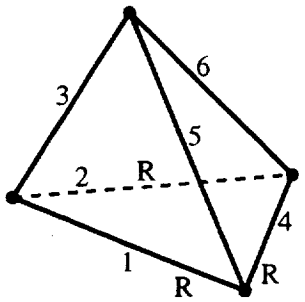



Fig. 2. Three types of subdivision are permitted for a tetrahedral element.

Mesh refinement is performed by first setting a bit flag to one for each edge that is targeted for subdivision. The edge markings for each element are then combined to form a 6-bit pattern as shown in Fig. 3 where the edges marked with an 'R' are the ones to be bisected. Elements are continuously upgraded to valid patterns corresponding to the three allowed subdivision types (cf. Fig. 2) until none of the patterns show any change. Once this edge marking is completed, each element is independently subdivided into smaller child elements based on its binary pattern. Special data structures are used to ensure that this process is computationally efficient.




| 6 5 4 3 2 1 | Edge number |
|---|---|
| 0 0 1 0 1 1 | Pattern = 11 |

Fig. 3. Sample edge-marking pattern for element subdivision.

Mesh coarsening also uses the edge-marking patterns. If a child element has any edge marked for coarsening, this element and its siblings are removed and their parent is reinstated. Parent edges and elements are retained at each refinement step so they do not have to be reconstructed. Reinstated parent elements have their edge-marking patterns adjusted to reflect that some edges have been coarsened. The parents are then subdivided based on their new patterns by invoking the mesh refinement procedure. As a result, the coarsening and refinement procedures share much of the same logic.

There are some constraints for mesh coarsening. For example, edges cannot be coarsened beyond the initial mesh. Edges must also be coarsened in an order that is reversed from the one by which they were refined. Moreover, an edge can coarsen if and only if its sibling is also targeted for coarsening. More details about these coarsening constraints are given in [8].

Details of the data structures are given in [7]; however, a brief description of the salient features is necessary to understand the distributed-memory implementation of the mesh adaptation code. Pertinent information is maintained for the vertices, elements, edges, and boundary faces of the mesh. For each vertex, the coordinates are stored in `coord[3]`, the solution in `soln[5]`, and a pointer to the first entry in the edge list in `edges`. The edge list for a vertex is a linked list of pointers to all the edges that are incident upon it. Such lists eliminate extensive searches and are crucial to the efficiency of the overall adaptation scheme. The tetrahedral elements have their six edges stored in `tedge[6]`, the edge-marking pattern in `patt`, the parent element in `tparent`, and the first child element in `tchild`. Sibling elements always reside contiguously in memory; hence, a parent element only needs a pointer to the first child. For each edge, we store its two end-points in `vertex[2]`, its parent edge in `eparent`, its two children edges in `echild[2]`, the two boundary faces it defines in `bfac[2]`, and a pointer to the first entry in the element list in `elems`. The element list for an edge is again a linked list of pointers to all the elements that share it. Finally, for each boundary face, we store the three edges in `bedge[3]`, the element to which it belongs in `belem`, the parent in `bparent`, and the first child in `bchild`. Sibling boundary faces, like elements, are stored consecutively in memory.

### 2.2 *Parallel Implementation*

The parallel version of the 3D_TAG mesh adaptation code contains an additional 3,000 lines of C++ with Message-Passing Interface (MPI), allowing portability to any system supporting these languages. This code is a wrapper around the original mesh adaptation program written in C, and required the addition of only 10 instructions to link it with the parallel constructs. The object-oriented approach allowed us to build a clean interface between the two layers of the program while maintaining efficiency. Only a slight increase in space was necessary to keep track of the global mappings and shared processor lists (SPLs) for objects on partition boundaries.

Parallel 3D_TAG consists of three phases: initialization, execution, and finalization. The initialization step consists of scattering the global data across the processors, defining a local numbering scheme for each object, and creating the mapping for objects that are shared by multiple processors. The execution step runs a copy of 3D_TAG on each processor that refines or coarsens its local region, while maintaining a globally-consistent grid along partition boundaries. Parallel performance is extremely critical during this phase since it will be executed several times during a computation. Finally, a gather operation is performed in the finalization step to combine the local grids into one global mesh. Locally-numbered objects and the corresponding pointers are reordered to represent one single consistent mesh. Note from Fig. 1 that the initialization and finalization phases are invoked only once for each problem outside the solution$\leftrightarrow$execution$\leftrightarrow$load-balancing cycle.

In order to perform parallel mesh adaptation, the initial grid must first be partitioned among the available processors. A good partitioner minimizes the total execution time by equidistributing the workload and reducing the interprocessor communication. However, it is also important within our framework that the partitioning phase be performed rapidly. There are several excellent heuristic algorithms for solving the NP-hard graph partitioning problem [20,28,29,32,34]. We used the ParMETIS [19] parallel multilevel partitioning algorithm for the test cases in this paper. ParMETIS reduces the size of the graph by collapsing vertices and edges using a heavy edge matching scheme, applies a greedy graph growing algorithm for partitioning the coarsest graph, and then uncoarsens it back using a combination of boundary greedy and Kernighan-Lin refinement to construct a partitioning for the original graph.

#### 2.2.1 *Initialization*

The initialization phase takes as input the global initial grid and the corresponding partitioning that maps each tetrahedral element to exactly one

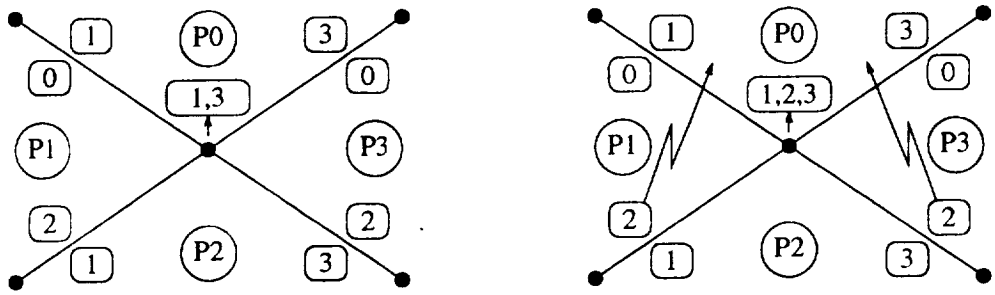
partition. The element data and partition information are then broadcast to all processors which, in parallel, assign a local, zero-based natural number to each element. We are thus assuming that an initial tetrahedral mesh exists, and that it is partitioned among the available processors. Once the elements have been processed, local edge information can be computed.

In three dimensions, an individual edge may belong to an arbitrary number of elements. Since each element is assigned to only one partition, it is theoretically possible for an edge to be shared by all the processors. For each partition, a local zero-based natural number is assigned to every edge that belongs to at least one element. Each processor then redefines its elements in `tedge[6]` in terms of these local edge numbers. Edges that are shared by more than one processor are identified by searching for elements that lie on partition boundaries. A bit flag is set to distinguish between shared and internal edges. A SPL is also generated for each shared edge. Finally, the element list in `elems` for each edge is updated to contain only the local elements.

The vertices are initialized using the `vertex[2]` data structure for each edge. Every local vertex is assigned a zero-based natural number in each partition. Next the local edge list for each vertex is created from the appropriate subset of the global `edges` array. Like shared edges, each shared vertex must be identified and assigned its SPL. A naive approach would be to thread through the data structures to the elements and their partitions to determine which vertices lie on partition boundaries. But this procedure requires excessive indirection. A faster approach is based on the following two properties of a shared vertex: it must be an end-point for at least one shared edge, and its SPL is the union of its shared edges' SPLs. However, some communication is required when using this method. For each vertex containing a shared edge in its `edges` list, that edge's SPL is communicated to the processors in the SPLs of all other shared edges until the union of all the SPLs is formed. For the cases in this paper, this process required no more than three iterations, and all shared vertices were processed as a function of the number of shared edges plus a small communication overhead. An example is shown in Fig. 4 where the SPL is being formed in P0 for the center vertex that is shared by three other processors. Without communication, P0 would incorrectly conclude that the vertex is shared only with P1 and P3.

The final step in the initialization phase is the local renumbering of the external boundary faces[5]. Since a boundary face belongs to only one element, it is never shared among processors. Each boundary face is defined by its three edges in `bedge[3]`, while each edge maintains a pair of pointers in `bfac[2]` to the boundary faces it defines. Since the global mesh is closed (water-tight), an edge on the external boundary is shared by exactly two boundary faces.

[5] The internal faces are not stored in the mesh data structures.

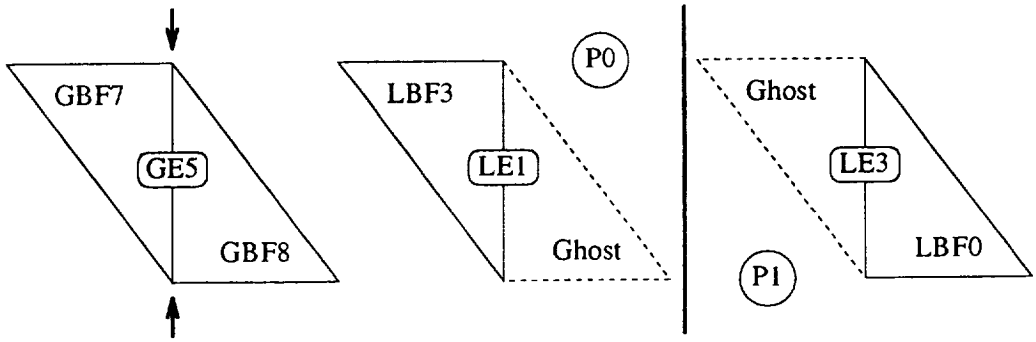



Before communication
P0 shares center vertex with P1, P3

After communication
P0 shares center vertex with P1, P2, P3

Fig. 4. An example showing the communication need to form the SPL for a shared vertex.

However, when the mesh is partitioned, this is no longer true. An example is shown in Fig. 5. An affected edge creates an empty ghost boundary face in each of the two processors for the execution phase. The ghost boundary faces do not participate in the adaptation process but are required to create a valid subgrid in each processor. These ghost faces are later eliminated during the finalization stage.




Before partitioning
Global edge GE5 shared by
global bdy faces GBF7 and GBF8

After partitioning
GE5 stored as LE1 and LE3 in P0 and P1
GBF7 as LBF3 in P0; GBF8 as LBF0 in P1

Fig. 5. An example showing how boundary faces are represented at partition boundaries.

A new data structure has been added to the serial code to represent all this shared information. Each shared edge and vertex contains a two-way mapping between its local and its global numbers[6], and a SPL of processors where its shared copies reside. The maximum additional storage depends on the number of processors used and the fraction of shared objects. For the cases in this paper, this was less than 10% of the memory requirements of the serial version.

[6] The global numbers for the various mesh objects are obtained trivially during the initialization phase.

### 2.2.2 *Execution*

The first step in the actual mesh adaptation phase is to target edges for refinement or coarsening. This is usually based on an error indicator for each edge that is computed from the solution. This strategy results in a symmetrical marking of all shared edges across partitions since such edges have the same numerical and geometrical information regardless of their processor number. However, elements have to be continuously upgraded to one of the three allowed subdivision patterns shown in Fig. 2. This causes some propagation of edges being targeted that could mark local copies of shared edges inconsistently. This is because the local geometry and marking patterns affect the nature of the propagation. Communication is therefore required after each iteration of the propagation process. Every processor sends a list of all the newly-marked local copies of shared edges to all the other processors in their SPLs. This process may continue for several iterations, and edge markings could propagate back and forth across partitions.

Figure 6 shows a two-dimensional example of two iterations of the propagation process across a partition boundary. The process is similar in three dimensions. Processor P0 marks its local copy of shared edge GE1 and communicates that to P1. P1 then marks its own copy of GE1, which causes some internal propagation because element marking patterns must be upgraded to those that are valid. Note that P1 marks its third internal edge and its local copy of shared edge GE2 during this phase. Marking information about GE2 is then communicated to P0, and the propagation phase terminates. The four original triangles can now be correctly subdivided into a total of 12 smaller triangles.

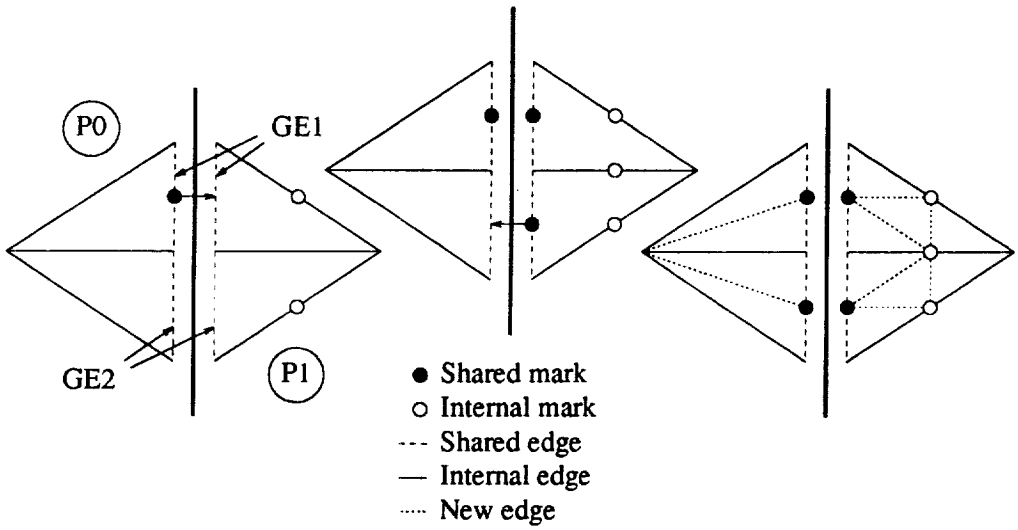



Fig. 6. A two-dimensional example showing communication during propagation of the edge marking phase.

Once all edge markings are complete, each processor executes the mesh adaptation code without the need for further communication, since all edges are

consistently marked. The only task remaining is to update the shared edge and vertex information as the mesh is adapted. This is handled as a post-processing phase.

New edges and vertices that are created during refinement are assigned shared processor information that depends on several factors. Four different cases can occur when new edges are created:

- If an *internal* edge is bisected, the center vertex and all new edges incident on that vertex are also internal to the partition. Shared processor information is not required in this case.
- If a *shared* edge is bisected, its two children and the center vertex inherit its SPL, since they lie on the same partition boundary.
- If a new edge is created in the *interior* of an element, it is internal to the partition since processor boundaries only lie along element faces. Shared processor information is not required.
- If a new edge is created that lies *across an element face*, communication is required to determine whether it is shared or internal. If it is shared, the SPL must be formed.

All the cases are straightforward, except for the last one. If the intersection of the SPLs of the two end-points of the new edge is null, the edge is internal. Otherwise, communication is required with the shared processors to determine whether they have a local copy of the edge. This communication is necessary because no information is stored about the internal faces of the tetrahedral elements. An alternate solution would be to incorporate internal faces as an additional object into the data structures, and maintaining it through the adaptation. However, this strategy does not compare favorably in terms of memory or CPU time to a single communication at the end of the refinement procedure. This is primarily because the number of triangular faces for a tetrahedral mesh is asymptotically ten times the number of mesh vertices.

Figure 7 shows the top view of a tetrahedron in processor P0 that shares two faces with P1 while the third face is internal. The fourth face is not shown and is irrelevant for this example. Assume that due to mesh refinement, three new edges LE1, LE2, and LE3, are formed in P0. An intersection of the SPLs for the two end-points of all the three edges yields P1. However, when P0 communicates this information to P1, P1 will only have local copies corresponding to LE1 and LE2. Thus, P0 would be able to correctly classify LE1 and LE2 as shared edges but LE3 as an internal edge.

The coarsening phase purges the data structures of all edges that are removed, as well as their associated vertices, elements, and boundary faces. No new shared processor information is generated since no mesh objects are created during this step. However, objects are renumbered as a result of compaction

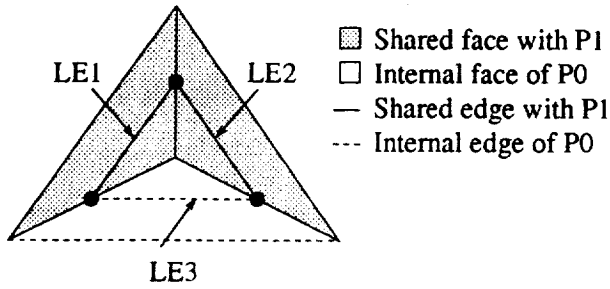



Fig. 7. An example showing how a new edge across a face is classified as shared or internal.

and all internal and shared data are updated accordingly. The refinement routine is then invoked to generate a valid mesh from the vertices left after the coarsening.

### *2.2.3 Finalization*

Under certain conditions, it is necessary to create a single global mesh after one or more adaptation steps. Some post processing tasks, such as visualization, need to process the whole grid simultaneously. Storing a snapshot of a grid for future restarts could also require a global view. Our finalization phase accomplishes this goal by merging the individual subgrids into one global data structure.

Each local object is first assigned a unique global number. Next, all local data structures are updated in terms of these global numbers. Finally, gather operations are performed to a host processor to create the global mesh. Individual processors are responsible for correctly arranging the data so that the host only collects and concatenates without further processing.

It is relatively simple to assign global element numbers since elements are not shared among processors. By performing a scan-reduce add[7] on the total number of elements, each processor can assign the final global element number. The global boundary face numbering is also done similarly since they too are not shared among processors.

Assigning global numbers to edges and vertices is somewhat more complicated since they may be shared by several processors. Each shared edge (and vertex) is assigned an owner from its SPL which is then responsible for generating the global number. Owners are randomly selected to keep the computation and communication loads balanced. Once all processors complete numbering their edges (and vertices), a communication phase propagates the global values from owners to other processors that have local copies.

[7] A *scan-reduce add* operation creates a vector whose $i$th element is the addition of the first $i-1$ elements of the argument vector.

After global numbers have been assigned to every object, all data structures are updated to contain consistent global information. Since elements and boundary faces are unique in each processor, no duplicates exist. All unowned edge copies are removed from the data structures, which are then compacted. However, the element lists in `elems` cannot be discarded for the unowned edges. Some communication is required to adjust the pointers in the local lists so that global lists can be formed without any serial computation. The pair of pointers in `bfac[2]` that were split during the initialization phase for shared edges are glued back by communicating the boundary face information to the owner. Vertex data structures are updated much like edges except for the manner in which their edge lists in `edges` are handled. Since shared vertices may contain local copies of the same global edge in their lists on different processors, the unowned edge copies are first deleted. Pointers are next adjusted as in the `elems` case with some communication among processors.

At this time, all processors have updated their local data with respect to their relative positions in the final global data structures. A gather operation by a host processor is performed to concatenate the local data structures. The host can then interface the global mesh directly to the appropriate post-processing module without having to perform any serial computation.

## 3 Dynamic Load Balancing

PLUM [23] is a novel method to dynamically balance the processor workloads for unstructured adaptive-grid computations with a global view. It has five novel features:

- Repeated use of the initial mesh dual graph keeps the connectivity and partitioning complexity constant during the course of an adaptive computation.
- Parallel mesh repartitioning avoids a potential serial bottleneck.
- Fast heuristic remapping assigns partitions to processors so that the redistribution cost is minimized.
- Efficient data movement significantly reduces the cost of remapping and mesh subdivision.
- Accurate metrics estimate and compare the computational gain and the redistribution cost of having a balanced workload after each mesh adaptation.

### *3.1 Repartitioning the Initial Mesh Dual Graph*

Repeatedly using the dual of the initial computational mesh for dynamic load balancing is one of the key features of PLUM. Each dual graph vertex has

two weights associated with it. The computational weight, $w_{\text{comp}}$, models the workload for the corresponding element. The remapping weight, $w_{\text{remap}}$, models the cost of moving the element from one processor to another. Every edge of the dual graph also has a weight, $w_{\text{comm}}$, that models the runtime interprocessor communication. These three weights are determined by the numerical algorithm and the data structures. In our current work, $w_{\text{comp}}$ is set to the number of leaf elements in the refinement tree, $w_{\text{remap}}$ is set to the total number of elements in the refinement tree, and $w_{\text{comm}}$ is set to the number of faces in the computational mesh that corresponds to the dual graph edge. The mesh connectivity, $w_{\text{comp}}$, and $w_{\text{comm}}$, together determine how balanced partitions with minimum runtime communication are formed. The $w_{\text{remap}}$ determine how partitions should be assigned to processors such that the data redistribution cost is minimized. New computational grids obtained by hierarchical adaptation are translated to $w_{\text{comp}}$ and $w_{\text{remap}}$ for every vertex and to $w_{\text{comm}}$ for every edge in the dual mesh. If the dual graph with a new set of weights is deemed unbalanced, the mesh is repartitioned using the ParMETIS [19] parallel multilevel partitioner.

### *3.2 Processor Reassignment*

New partitions generated by a partitioner must be mapped to processors such that the data redistribution cost is minimized. In general, the number of new partitions is an integer multiple $F$ of the number of processors, and each processor is assigned $F$ unique partitions. Allowing multiple partitions per processor reduces the volume of data movement at the expense of partitioning and processor reassignment times [23]; however, the simpler scheme of setting $F$ to unity suffices for most practical applications.

We first generate a similarity measure $M$ that indicates how the remapping weights $w_{\text{remap}}$ of the new partitions are distributed over the processors. It is represented as a matrix where entry $M_{ij}$ is the sum of the $w_{\text{remap}}$ values of all the dual graph vertices in new partition $j$ that already reside on processor $i$. Various cost functions are usually needed to solve the processor reassignment problem using $M$ for different machine architectures. We present three general metrics: `TotalV`, `MaxV`, and `MaxSR`, which model the remapping cost on most multiprocessor systems. `TotalV` minimizes the total volume of data moved among all the processors, `MaxV` minimizes the maximum flow of data to *or* from any single processor, while `MaxSR` minimizes the sum of the maximum flow of data to *and* from any processor. A greedy heuristic algorithm to minimize the remapping overhead is also presented.

### 3.2.1 *TotalV Metric*

The `TotalV` metric assumes that by reducing network contention and the total number of elements moved, the remapping time will be reduced. In general, each processor cannot be assigned $F$ unique partitions corresponding to their $F$ largest weights. To minimize `TotalV`, each processor $i$ must be assigned $F$ partitions $j_{i\_f}$, $f = 1, 2, \ldots, F$, such that the objective

$$\sum_{i=1}^{P} \sum_{f=1}^{F} M_{ij_{i\_f}}$$

is maximized subject to the constraint

$$j_{i\_r} \neq j_{k\_s}, \text{ for } i \neq k \text{ or } r \neq s; \quad i, k = 1, 2, \ldots, P; \quad r, s = 1, 2, \ldots, F.$$

We can optimally solve this by mapping it to a network flow optimization problem described as follows. Let $G = (V, E)$ be an undirected graph. $G$ is *bipartite* if $V$ can be partitioned into two sets $A$ and $B$ such that every edge has one vertex in $A$ and the other vertex in $B$. A *matching* is a subset of edges, no two of which share a common vertex. A *maximum-cardinality matching* is one that contains as many edges as possible. If $G$ has a real-valued cost on each edge, we can consider the problem of finding a maximum-cardinality matching whose total edge cost is maximized. We refer to this as the *maximally weighted bipartite graph* (MWBG) problem (also known as the *assignment* problem).

When $F = 1$, optimally solving for the `TotalV` metric trivially reduces to MWBG, where $V$ consists of $P$ processors and $P$ partitions in each set. An edge of weight $M_{ij}$ exists between vertex $i$ of the first set and vertex $j$ of the second set. If $F > 1$, the processor reassignment problem can be reduced to MWBG by duplicating each processor and all of its incident edges $F$ times. Each set of the bipartite graph then has $P \times F$ vertices. After the optimal solution is obtained, the solutions for all $F$ copies of a processor are combined to form a one-to-$F$ mapping between the processors and the partitions. The optimal solution for the `TotalV` metric and the corresponding processor assignment of an example similarity matrix is shown in Fig. 8(a).

The fastest MWBG algorithm [13] can compute a matching in $O(|V|^2 \log |V| + |V||E|)$ time, or in $O(|V|^{1/2}|E| \log(|V|C))$ time if all edge costs are integers of absolute value at most $C$ [15]. We have implemented the optimal algorithm with a runtime of $O(|V|^3)$. Since $M$ is generally dense, $|E| \approx |V|^2$, implying that we should not see a dramatic performance gain from a faster implementation.

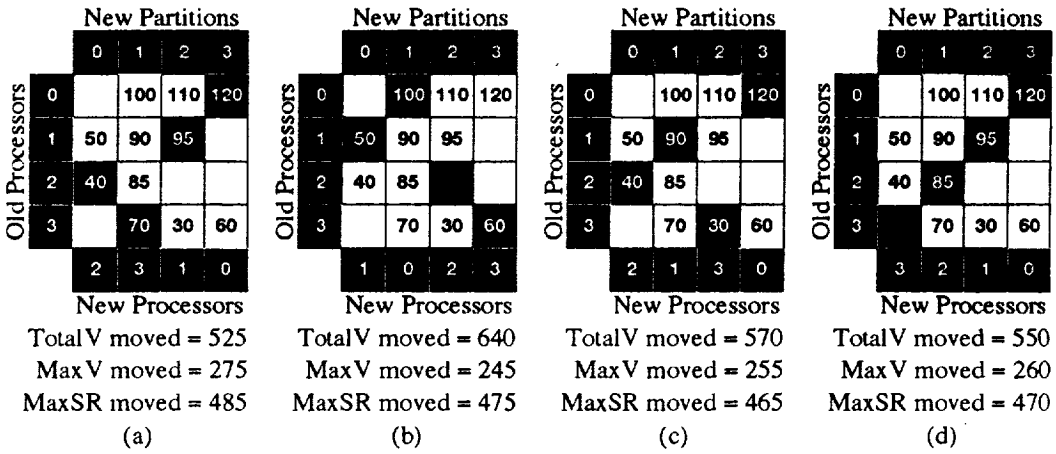



Fig. 8. Various cost metrics of a similarity matrix $M$ for $P = 4$ and $F = 1$ using (a) the optimal MWBG, (b) the optimal BMCM, (c) the optimal DBMCM, and (d) our heuristic algorithms.

### *3.2.2 MaxV Metric*

The metric `MaxV`, unlike `TotalV`, considers data redistribution in terms of solving a load imbalance problem, where it is more important to minimize the workload of the most heavily-weighted processor than to minimize the sum of all the loads. During the process of remapping, each processor must pack and unpack send and receive buffers, incur remote-memory latency time, and perform the computational overhead of rebuilding internal and shared data structures. By minimizing $\max(\alpha \times \max(\texttt{ElemsSent}), \beta \times \max(\texttt{ElemsRecd}))$, where $\alpha$ and $\beta$ are machine-specific parameters, `MaxV` attempts to reduce the total remapping time by minimizing the execution time of the most heavily-loaded processor. We can solve this optimally by considering the problem of finding a maximum-cardinality matching whose maximum edge cost is minimum. We refer to this as the *bottleneck maximum cardinality matching* (BMCM) problem.

To find the BMCM of the graph $G$ corresponding to the similarity matrix, we first need to transform $M$ into a new matrix $M'$. Each entry $M'_{ij}$ represents the maximum cost of migrating data between processor $i$ and partition $j$:

$$M'_{ij} = \max\left((\alpha \sum_{y=1}^{P} M_{iy}, y \neq j), (\beta \sum_{x=1}^{P} M_{xj}, x \neq i)\right).$$

Optimally solving the BMCM problem is NP-complete for $F > 1$. For $F = 2$, it is NP-complete by reduction from numerical matching with target sums; for $F > 2$, it is NP-complete by reduction from 3-partition. We have implemented the BMCM algorithm in [3] for $F = 1$ which combines a maximum cardinality matching algorithm with a binary search, and runs in $O(|V|^{1/2}|E| \log |V|)$. The fastest known BMCM algorithm [14] has a runtime of $O((|V| \log |V|)^{1/2}|E|)$.

The new processor assignment for the similarity matrix in Fig. 8 using this approach with $\alpha = \beta = 1$ is shown in Fig. 8(b). Notice that the total number of elements moved in Fig. 8(b) is larger than the corresponding value in Fig. 8(a); however, the maximum number of elements moved is smaller.

### *3.2.3 MaxSR Metric*

Our third metric, **MaxSR**, is similar to **MaxV** in the sense that the overhead of the bottleneck processor is minimized during the remapping phase. **MaxSR** differs, however, in that it minimizes the sum of the heaviest data flow from any processor *and* to any processor, expressed as ($\alpha \times$ max (**ElemsSent**) + $\beta \times$ max (**ElemsRecd**)). We refer to this as the *double* bottleneck maximum cardinality matching (DBMCM) problem. The **MaxSR** formulation allows us to capture the computational overhead of packing and unpacking data, when these two phases are separated by a barrier synchronization. Additionally, the **MaxSR** metric may also approximate the many-to-many communication pattern of our remapping phase. Since a processor can either be sending or receiving data, the overhead of these two phases should be modeled as a sum of costs.

We have developed an algorithm for computing the minimum **MaxSR** of the graph $G$ corresponding to our similarity matrix. We first transform $M$ to a new matrix $M''$. Each entry $M''_{ij}$ contains a pair of values $\{S_{ij}, R_{ij}\}$ corresponding to the total cost of sending and receiving data, when partition $j$ is mapped to processor $i$:

$$M''_{ij} = \left\{ S_{ij} = (\alpha \sum_{y=1}^{P} M_{iy}, y \neq j)\ ,\ R_{ij} = (\beta \sum_{x=1}^{P} M_{xj}, x \neq i) \right\}.$$

The optimal algorithm for the **MaxSR** metric is NP-complete for $F > 1$, since the underlying BMCM algorithm is also NP-complete.

Let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be the distinct $S_{ij}$ values appearing in $M''$, sorted in increasing order. Thus, $\sigma_i < \sigma_{i+1}$ and $k \leq P^2$. Form the bipartite graph $G_i = (V, E_i)$, where $V$ consists of processor vertices $u = 1, 2, \ldots, P$ and partition vertices $v = 1, 2, \ldots, P$, and $E_i$ contains edge $(u, v)$ if $S_{uv} \leq \sigma_i$; furthermore, edge $(u, v)$ has weight $R_{uv}$ if it is in $E_i$.

For small values of $i$, graph $G_i$ may not have a perfect matching. Let $i_{\min}$ be the smallest index such that $G_{i_{\min}}$ has a perfect matching. Obviously, $G_i$ has a perfect matching for all $i \geq i_{\min}$. Solving the BMCM problem of $G_i$ gives a matching that minimizes the maximum $R_{ij}$ edge weight. It gives a matching

with **MaxSR** value at most $\sigma_i + \mathbf{MaxV}(G_i)$. Defining

$$\mathbf{MaxSR}(i) = \min_{i_{\min} \le j \le i} (\sigma_j + \mathbf{MaxV}(G_j)),$$

it is easy to see that $\mathbf{MaxSR}(k)$ equals the correct value of **MaxSR**. Thus, our algorithm computes **MaxSR** by solving $k$ BMCM problems on the graphs $G_i$ and computing the minimum value $\mathbf{MaxSR}(k)$. However, we can prematurely terminate the algorithm if there exists an $i_{\max}$ such that $\sigma_{i_{\max}+1} \ge \mathbf{MaxSR}(i_{\max})$, since it is then guaranteed that the **MaxSR** solution is $\mathbf{MaxSR}(i_{\max})$.

Our implementation has a runtime of $O(|V|^{1/2}|E|^2 \log |V|)$ since the BMCM algorithm is called $|E|$ times in the worst case; however, it can be decreased to $O(|E|^2)$. The following is a sketch of this more efficient implementation.

Suppose we have constructed a matching $\mu$ that solves the BMCM problem of $G_i$ for $i \ge i_{\min}$. We solve the BMCM problem of $G_{i+1}$ as follows. Initialize a working graph $G$ to be $G_{i+1}$ with all edges of weight greater than $\mathbf{MaxV}(G_i)$ deleted. Take the matching $\mu$ on $G$, and delete all unmatched edges of weight $\mathbf{MaxV}(G_i)$. Choose an edge $(u, v)$ of maximum weight in $\mu$, remove it from $\mu$ and $G$, and search for an augmenting path from $u$ to $v$ in $G$. If no such path exists, we know that $\mathbf{MaxV}(G_i) = \mathbf{MaxV}(G_{i+1})$. If an augmenting path is found, repeat this procedure by choosing a new edge $(u', v')$ of maximum weight in the matching and searching for an augmenting path. After some repetitions of this procedure, the maximum weight of a matched edge will have decreased to the desired value $\mathbf{MaxV}(G_{i+1})$. At this point our algorithm to solve the BMCM problem of $G_{i+1}$ will stop, since no augmenting path will be found.

This algorithm is of complexity $O(|E|^2)$ since each search for an augmenting path uses $O(|E|)$ time and there are $O(|E|)$ such searches. A successful search for an augmenting path for edge $(u, v)$ permanently eliminates it from all future graphs, so there are at most $|E|$ successful searches. Furthermore, there are at most $|E|$ unsuccessful searches, one for each value of $i$.

The new processor assignment for the similarity matrix in Fig. 8 using the DBMCM algorithm with $\alpha = \beta = 1$ is shown in Fig. 8(c). Notice that the **MaxSR** solution is minimized; however, the number of **TotalV** elements moved is larger than the corresponding value in Fig. 8(a), and more **MaxV** elements are moved than in Fig. 8(b). Also note that the optimal similarity matrix solution for **MaxSR** is provably no more than twice that of **MaxV**.

### *3.2.4 Heuristic Algorithm*

We have developed a heuristic greedy algorithm that gives a suboptimal solution to the **TotalV** metric in $O(|E|)$ steps [23]. All partitions are initially

flagged as unassigned and each processor has a counter set to $F$ that indicates the remaining number of partitions it needs. The non-zero entries of the similarity matrix $M$ are then sorted in descending order. Starting from the largest entry, partitions are assigned to processors that have less than $F$ partitions until done. If necessary, the zero entries in $M$ are also used. It has been proven that a processor assignment obtained using the heuristic algorithm can never result in a data movement cost that is more than twice that of the optimal `TotalV` assignment [23]. In addition, experimental results in § 4.3 demonstrate that our heuristic quickly finds high quality solutions for all three metrics. Applying this heuristic algorithm to the similarity matrix in Fig. 8 generates the new processor assignment shown in Fig. 8(d).

### *3.3 Remapping Cost Model*

After the new partitions are reassigned to the processors, a model is required to predict the redistribution cost for a given machine. Accurately estimating this time is difficult because of the number and complexity of the costs involved in the remapping procedure. The total remapping cost includes the computational overhead for rebuilding internal data structures and updating shared boundary information. The communication overhead is architecture-dependent and complicated because of the many-to-many collective communication pattern used by the remapper.

Our redistribution algorithm first removes the data objects moving out of a partition and places them in a buffer. A collective communication then appropriately distributes the data to their final destination, where they are integrated into the data structures. Finally, the partition boundary information is consistently updated. This remapping strategy closely follows the superstep model of BSP [31].

The expected redistribution time on bandwidth-rich systems is then given by:

$$\gamma \times \texttt{MaxSR} + O,$$

where $\texttt{MaxSR} = \max(\texttt{ElemsSent}) + \max(\texttt{ElemsRecd})$, $\gamma$ is the total computation and communication cost to process each redistributed element, and $O$ is the predicted sum of all constant overheads [23]. This formulation demonstrates the need to model and minimize the `MaxSR` metric when performing processor reassignment. To compute the values of $\gamma$ and $O$, a simple least squares fit through several data points for various redistribution patterns and their corresponding runtimes can be used. This procedure needs to be performed only once for each architecture, and the values of $\gamma$ and $O$ can then be used in actual computations to estimate the redistribution cost.

## 4 Experimental Results

The parallel 3D_TAG mesh adaptation procedure and the PLUM global load balancing strategy have been implemented in C and C++, with the parallel activities in MPI for portability. All experiments were performed on the wide-node SP2 at NASA Ames, the Origin2000 at NCSA, and the T3E at NASA Goddard, without any machine-specific optimizations.

Our computational mesh is the one used to simulate the acoustics experiment of Purcell [24] where a 1/7th-scale model of a UH-1H helicopter rotor blade was tested over a range of Mach numbers. Detailed numerical results of the simulation are given elsewhere [30]. This paper reports only on the performance of parallel 3D_TAG and PLUM.

Performance results are presented for one refinement and one coarsening step using various edge-marking strategies. Six strategies are used for the refinement step. The first set of experiments, denoted as RAND_1R, RAND_2R, and RAND_3R, consists of randomly bisecting 5%, 33%, and 60% of the edges in the mesh, respectively. The second set, denoted as REAL_1R, REAL_2R, and REAL_3R, consists of bisecting the same numbers of edges using an error indicator [30] derived from the actual solution. These strategies represent significantly different scenarios. In practice, mesh adaptation tends to be local. The RAND cases are included as they are expected to behave somewhat ideally because the computational loads are automatically balanced. Thus, the RAND results should give an indirect indication of how well parallel 3D_TAG can really perform without explicit load balancing.

Since the coarsening procedure and performance are similar to the refinement method, only two cases are presented where 7% of the edges in the refined meshes obtained with the RAND_2R and the REAL_2R strategies are respectively coarsened randomly (RAND_2C) or based on actual solution (REAL_2C). Table 1 presents the progression of grid sizes through the two adaptation steps for each edge-marking strategy.

### *4.1 Refinement Phase*

Table 2 presents the computation times and parallel speedup for the refinement step with the random marking of edges (strategies RAND_1R, RAND_2R, and RAND_3R). Note that the speedup values are calculated based on the total time. Performance is excellent with efficiencies of more than 83% on 32 processors and 76% on 64 processors for the RAND_3R case. Parallel mesh refinement shows a markedly better performance for RAND_3R due to its bigger computation-to-communication ratio. In general, the total speedup will

Table 1
Grid sizes for the different refinement and coarsening strategies

| | Vertices | Elements | Edges | Bdy Faces |
|---|---|---|---|---|
| Initial mesh | 13,967 | 60,968 | 78,343 | 6,818 |
| RAND_1R | 18,274 | 82,417 | 104,526 | 7,672 |
| REAL_1R | 17,880 | 82,489 | 104,209 | 7,682 |
| RAND_2R | 39,829 | 201,734 | 246,949 | 10,774 |
| REAL_2R | 39,332 | 201,780 | 247,115 | 12,008 |
| RAND_3R | 60,916 | 320,919 | 389,686 | 15,704 |
| REAL_3R | 61,161 | 321,841 | 391,233 | 16,464 |
| RAND_2C | 21,756 | 100,537 | 126,448 | 8,312 |
| REAL_2C | 20,998 | 100,124 | 125,261 | 8,280 |

improve as the size of the refined mesh increases. This is because the mesh adaptation time will increase while the percentage of elements along processor boundaries will decrease.

Table 2
Performance of mesh refinement when edges are bisected randomly

| | | RAND_1R | | RAND_2R | | RAND_3R | |
|---|---|---|---|---|---|---|---|
| | Shared | Comp | Total | Comp | Total | Comp | Total |
| P | Edges | Time | Speedup | Time | Speedup | Time | Speedup |
| 1 | 0.0% | 7.044 | 1.00 | 26.904 | 1.00 | 45.015 | 1.00 |
| 2 | 1.9% | 3.837 | 1.84 | 13.878 | 1.94 | 22.762 | 1.98 |
| 4 | 3.7% | 2.025 | 3.48 | 7.605 | 3.54 | 11.569 | 3.89 |
| 8 | 6.6% | 1.068 | 6.58 | 4.042 | 6.65 | 5.913 | 7.61 |
| 16 | 8.8% | 0.587 | 11.86 | 2.293 | 11.67 | 3.191 | 14.07 |
| 32 | 11.6% | 0.330 | 20.72 | 1.338 | 19.78 | 1.678 | 26.62 |
| 64 | 15.3% | 0.191 | 32.92 | 0.711 | 35.82 | 0.896 | 48.66 |

The communication time is less than 3% of the total time for up to 32 processors for all three cases. On 64 processors, the communication time although still quite small, is more than 12% of the computation time for RAND_1R. This is because each of the 64 partitions contains less than 1,000 elements with more than 15% of the edges on partition boundaries. Since additional work and storage are necessary for shared edges, the speedup deteriorates as the percentage of such edges increases. The situation is much better for RAND_3R since the computation time is significantly higher.

Table 3 shows the computation times and speedup when edges are marked using a solution-based error indicator. Performance is extremely poor, especially for REAL_1R and REAL_2R, with speedups of only 9.2X and 19.2X on 64 processors, respectively. This is because mesh adaptation for practical problems occurs in a localized region, causing an almost worst case load-balance behavior. Elements are targeted for refinement on only a small subset of the available processors. Most of the processors remain idle since none of their assigned elements need to be refined. Performance is somewhat better for the REAL_3R strategy because the refinement region is much larger. Since 60% of all edges are bisected in this case, most of the processors are busy doing useful work. This is reflected by an efficiency of more than 56% on 64 processors.

Table 3
Performance of mesh refinement when edges are bisected based on actual solution

| | REAL_1R | | REAL_2R | | REAL_3R | |
|---|---|---|---|---|---|---|
| P | Comp Time | Total Speedup | Comp Time | Total Speedup | Comp Time | Total Speedup |
| 1 | 5.902 | 1.00 | 23.780 | 1.00 | 41.702 | 1.00 |
| 2 | 3.979 | 1.48 | 18.117 | 1.31 | 26.317 | 1.58 |
| 4 | 2.530 | 2.33 | 9.173 | 2.59 | 14.266 | 2.92 |
| 8 | 1.589 | 3.71 | 7.091 | 3.35 | 8.430 | 4.95 |
| 16 | 1.311 | 4.48 | 4.046 | 5.87 | 4.363 | 9.55 |
| 32 | 0.879 | 6.65 | 2.277 | 10.40 | 2.278 | 18.25 |
| 64 | 0.616 | 9.22 | 1.224 | 19.16 | 1.148 | 35.95 |

Note that the communication times constitute a much smaller fraction of the total time compared to the cases when edges are bisected randomly. This is due to the difference in the distribution of bisected edges. The RAND cases require significantly more communication among processors at the partition boundaries because refinement is scattered all over the problem domain. The REAL cases, on the other hand, require much less communication since the refined regions are localized and mostly contained within partitions.

Poor parallel performance of the mesh refinement code for the three REAL strategies is due to severe load imbalance. It is therefore worthwhile trying to load balance this phase of 3D_TAG as much as possible. This can be achieved within PLUM by splitting the mesh refinement step into two distinct phases of edge marking and mesh subdivision. After edges are marked for bisection, it is possible to exactly predict the new refined mesh before actually performing the subdivision phase. This is because elements are independently refined based on their binary patterns. The mesh is repartitioned if the edge markings are skewed beyond a specified tolerance. All necessary data is then appropriately

redistributed and the mesh elements are refined in their *destination* processors. This enables the subdivision phase to perform in a more load-balanced fashion. As a bonus, a smaller volume of data has to be moved around since remapping is performed before the mesh grows in size due to refinement.

Using this methodology, the three REAL cases were run again. Table 4 presents the performance results of this "load balanced" mesh refinement step. Compared to the results in Table 3, the parallel speedups are now much higher. In fact, the speedups for REAL_2R consistently beat the corresponding speedups for RAND_2R, while REAL_3R outperforms RAND_3R when more than eight processors are used. Even though the RAND cases are expected to behave somewhat ideally, these results show that explicit load balancing can do better. An efficiency of 82% is attained for REAL_3R on 64 processors, thereby demonstrating that mesh adaptation can deliver excellent speedups if the marked edges are well-distributed among the processors. Communication requires a larger fraction of the total time for this load balanced strategy because the mesh refinement work is distributed among more processors after load balancing. However, communication times are still relatively small, requiring less than 4% of the total time for all runs except for REAL_1R on 64 processors.

Table 4
Performance of "load balanced" mesh refinement

| | REAL_1R | | REAL_2R | | REAL_3R | |
|---|---|---|---|---|---|---|
| P | Comp Time | Total Speedup | Comp Time | Total Speedup | Comp Time | Total Speedup |
| 1 | 5.902 | 1.00 | 23.780 | 1.00 | 41.702 | 1.00 |
| 2 | 3.311 | 1.78 | 12.059 | 1.97 | 21.592 | 1.93 |
| 4 | 1.980 | 2.98 | 6.733 | 3.53 | 10.975 | 3.80 |
| 8 | 1.369 | 4.30 | 3.430 | 6.92 | 5.678 | 7.34 |
| 16 | 0.702 | 8.34 | 1.840 | 12.88 | 2.899 | 14.37 |
| 32 | 0.414 | 13.89 | 1.051 | 22.41 | 1.484 | 27.99 |
| 64 | 0.217 | 23.89 | 0.528 | 43.24 | 0.777 | 52.52 |

The effect of load balancing the refined mesh before performing the actual subdivision can be seen more directly from the results presented in Table 5 for RAND_3R and REAL_3R. The quality of load balance is defined as the ratio of the number of elements on the most heavily-loaded processor to the optimal number of elements per processor. For the RAND_3R strategy, the mesh was refined without any load balancing. Two different sets of results are presented for REAL_3R: one without load balancing (NLB) and the other using the technique of load balanced mesh refinement (LB). Notice that the quality of load balance before refinement is excellent, and identical, for both

Table 5
Quality of load balance before and after mesh refinement

| | RAND_3R | | NLB REAL_3R | | LB REAL_3R | |
|---|---|---|---|---|---|---|
| P | Before | After | Before | After | Before | After |
| 1 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| 2 | 1.000 | 1.016 | 1.000 | 1.556 | 1.406 | 1.000 |
| 4 | 1.000 | 1.033 | 1.000 | 2.188 | 1.948 | 1.000 |
| 8 | 1.000 | 1.085 | 1.000 | 6.347 | 2.654 | 1.000 |
| 16 | 1.000 | 1.167 | 1.000 | 5.591 | 4.025 | 1.000 |
| 32 | 1.001 | 1.226 | 1.001 | 7.987 | 4.212 | 1.000 |
| 64 | 1.005 | 1.506 | 1.005 | 8.034 | 6.709 | 1.004 |

RAND_3R and NLB REAL_3R because the initial mesh is partitioned using ParMETIS. However, after mesh refinement, the load imbalance is severe, particularly for NLB REAL_3R. The load imbalance is not too bad for RAND_3R since edges are randomly marked for refinement. This is reflected by the difference in the speedup values in Tables 2 and 3. For LB REAL_3R, the initial mesh is repartitioned after edge marking is complete. This imbalances the load before refinement, but generates partitions that are excellently balanced after subdivision is complete. It also improves the speedup values significantly.

### *4.2 Coarsening Phase*

The coarsening phase consists of three major steps: marking edges to coarsen, cleaning up all the data structures by removing the coarsened edges and their associated vertices and tetrahedral elements, and finally invoking the refinement routine to generate a valid mesh from the remaining vertices.

Timings and parallel speedup for the RAND_2C and the REAL_2C coarsening strategies are presented in Table 6. The follow-up mesh refinement times are not included because the goal was to demonstrate the parallel performance of only the modules that are required during the coarsening phase. The computation time in Table 6 is the time required to mark edges for coarsening. The communication time is negligible and not shown, but it was included when calculating the speedup values. The cleanup time, on the other hand, is always a significant fraction of the total time. The cleanup time decreases as more and more processors are used due to the reduction in the local mesh size for each individual partition; however, since it depends on the fraction of shared objects, performance deteriorates as the problem size is over-saturated by processors. For instance, even though the total efficiency is about 50% for

Table 6
Performance of mesh coarsening

| | RAND_2C | | | REAL_2C | | |
|---|---|---|---|---|---|---|
| P | Comp Time | Cleanup Time | Total Speedup | Comp Time | Cleanup Time | Total Speedup |
| 1 | 3.619 | 2.364 | 1.00 | 3.989 | 2.246 | 1.00 |
| 2 | 1.832 | 1.352 | 1.88 | 2.026 | 1.283 | 1.88 |
| 4 | 0.963 | 0.782 | 3.42 | 1.066 | 0.854 | 3.25 |
| 8 | 0.572 | 0.498 | 5.57 | 0.600 | 0.498 | 5.68 |
| 16 | 0.303 | 0.287 | 10.01 | 0.334 | 0.279 | 10.17 |
| 32 | 0.170 | 0.170 | 16.95 | 0.167 | 0.161 | 19.01 |
| 64 | 0.070 | 0.098 | 31.17 | 0.093 | 0.097 | 32.82 |

64 processors for the results in Table 6, the efficiency when considering only the cleanup times is barely 37%.

### *4.3 Comparison of Reassignment Algorithms*

Table 7 presents a comparison of our five different processor reassignment algorithms in terms of the reassignment time (in secs) and the amount of data movement. Results are shown for the REAL_2R strategy on the SP2 with $F = 1$. The ParMETIS [19] case does not require any explicit processor reassignment since we choose the default partition-to-processor mapping given by the partitioner. The poor performance is expected since ParMETIS is a global partitioner that does not attempt to minimize the remapping overhead. A detailed performance comparison of ParMETIS with other partitioners within the PLUM framework is given in [6].

The execution times of the other four algorithms increase with the number of processors because of the growth in the size of the similarity matrix; however, the time for the heuristic algorithm on 64 processors is still very small. The `TotalV`, `MaxV`, and `MaxSR` metrics are obviously minimized by the MWBG, BMCM, and DBMCM algorithms, respectively. All the algorithms almost match the minimum `MaxV` value given by BMCM. The extremely local refinement in our test case requires the migration of a large number of elements to achieve load balance, causing any reasonable reassignment algorithm to return a similar `MaxV` solution.

The DBMCM algorithm optimally reduces `MaxSR`, but achieves no more than a 5% improvement over the other algorithms. Nonetheless, since we believe

Table 7
Comparison of reassignment algorithms for REAL_2R on the SP2 with $F = 1$

| | P = 32 | | | | P = 64 | | | |
|---|---|---|---|---|---|---|---|---|
| | TotalV | MaxV | MaxSR | Reass. | TotalV | MaxV | MaxSR | Reass. |
| Algorithm | Metric | Metric | Metric | Time | Metric | Metric | Metric | Time |
| ParMETIS | 58,297 | 5,067 | 7,467 | 0.000 | 67,439 | 2,667 | 4,452 | 0.000 |
| MWBG | 34,738 | 4,410 | 5,822 | 0.018 | 38,059 | 2,261 | 3,142 | 0.065 |
| BMCM | 49,611 | 4,410 | 5,944 | 0.032 | 52,837 | 2,261 | 3,282 | 0.133 |
| DBMCM | 50,270 | 4,414 | 5,733 | 0.092 | 54,896 | 2,261 | 3,121 | 1.252 |
| Heuristic | 35,032 | 4,410 | 5,809 | 0.002 | 38,283 | 2,261 | 3,123 | 0.009 |

that the MaxSR metric can closely approximate the remapping cost on many architectures, computing its optimal solution can provide useful information. Notice that the minimum TotalV increases slightly as $P$ grows from 32 to 64, while MaxSR is dramatically reduced by over 45%. This trend continues as the number of processors increases, and indicates that PLUM will remain viable on a large number of processors, since the per processor workload decreases as $P$ increases.

Finally, observe that the heuristic algorithm does an excellent job in minimizing all three cost metrics in a trivial amount of time. Although theoretical bounds have only been established for the TotalV metric, empirical evidence indicates that the heuristic algorithm closely approximates both MaxV and MaxSR. Similar results were obtained for the other edge-marking strategies. Our heuristic algorithm has now been incorporated into ParMETIS [26]. This feature gives users the option of global repartitioning while minimizing the remapping overhead.

### *4.4 Portability Analysis*

The three left plots in Fig. 9 illustrate parallel speedup for the three edge-marking strategies on the SP2, Origin2000, and T3E. Two sets of results are presented for each machine: one when data remapping is performed after mesh refinement, and the other when remapping is performed before refinement. The REAL_3R case shows the best speedup values because it is the most computation intensive. Remapping data before refinement has the largest relative effect for REAL_1R, because it has the smallest refinement region and predictively load balancing the refined mesh returns the biggest benefit. The best results are for REAL_3R with remapping before refinement, showing an efficiency greater than 87% on 32 processors.

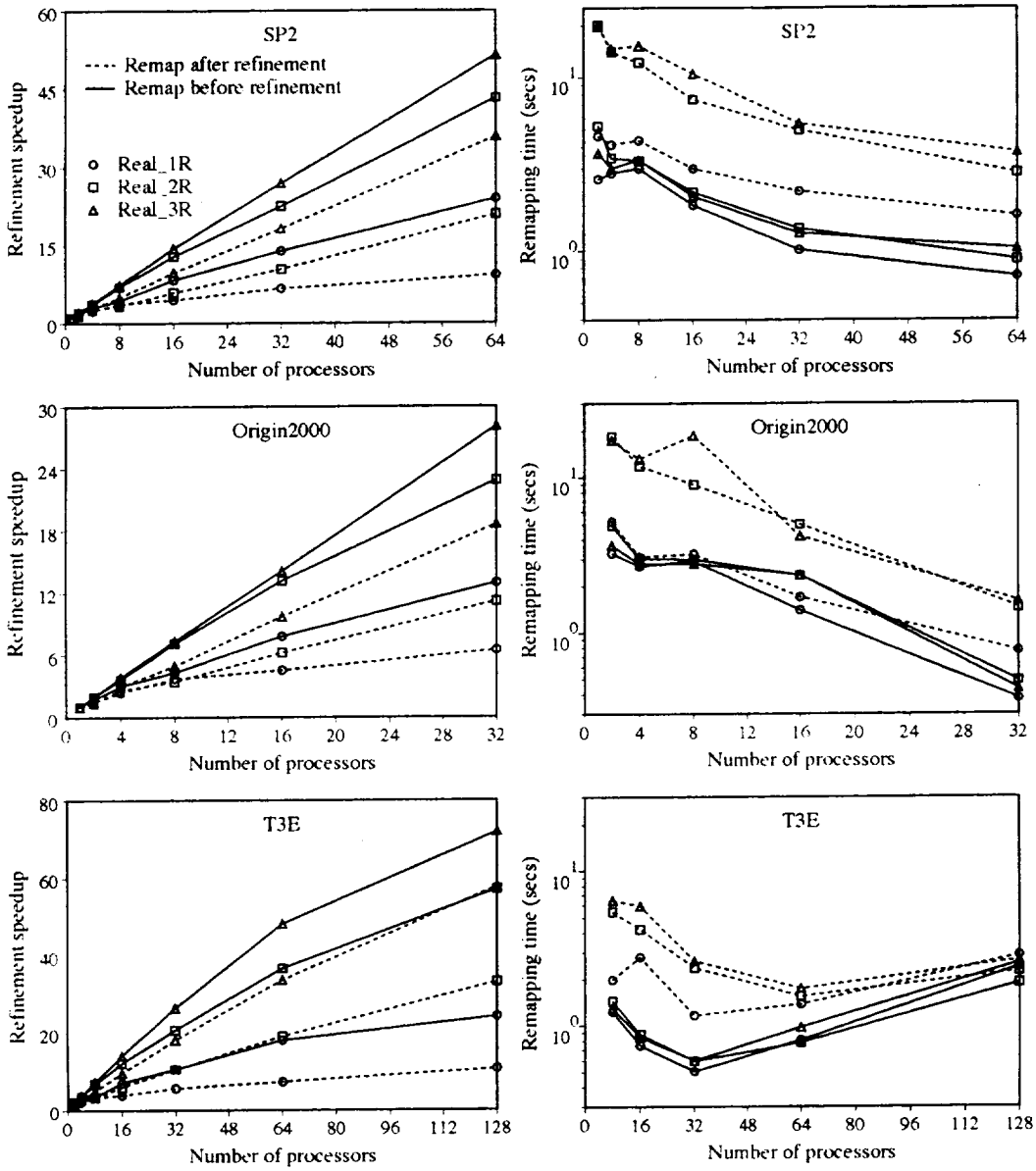



Fig. 9. Refinement speedup (left) and remapping time (right) within PLUM on the SP2, Origin2000, and T3E, when data is redistributed either after or before mesh refinement.

To compare the performance on the three target machines more critically, one needs to look at the actual times rather than the speedup values. Table 8 shows how the execution time (in secs) is spent during the refinement and subsequent load balancing phases for the REAL_2R case when data is remapped before the subdivision phase. Notice that the T3E adaptation times are consistently more than 1.4 times faster than the Origin2000 and three times faster than the SP2. One reason for this performance difference is the disparity in the clock speeds of the three machines. Another reason is that the mesh adaptation code does not use the floating-point units on the SP2, thereby adversely affecting its overall performance.

Table 8
Anatomy of execution times for REAL_2R on the Origin2000, SP2, and T3E

| | Adaptation Time | | | Remapping Time | | | Partitioning Time | | |
|---|---|---|---|---|---|---|---|---|---|
| P | O2000 | SP2 | T3E | O2000 | SP2 | T3E | O2000 | SP2 | T3E |
| 2 | 5.261 | 12.06 | 3.455 | 3.005 | 3.440 | 2.648 | 0.628 | 0.815 | 0.701 |
| 4 | 2.880 | 6.734 | 1.956 | 3.005 | 3.440 | 1.501 | 0.584 | 0.537 | 0.477 |
| 8 | 1.470 | 3.434 | 1.034 | 2.963 | 3.321 | 1.449 | 0.522 | 0.424 | 0.359 |
| 16 | 0.794 | 1.846 | 0.568 | 2.346 | 2.173 | 0.880 | 0.396 | 0.377 | 0.301 |
| 32 | 0.458 | 1.061 | 0.333 | 0.491 | 1.338 | 0.592 | 0.389 | 0.429 | 0.302 |
| 64 | | 0.550 | 0.188 | | 0.890 | 0.778 | | 0.574 | 0.425 |
| 128 | | | 0.121 | | | 1.894 | | | 0.599 |

The three right plots in Fig. 9 show the remapping time for each of the three cases on the SP2, Origin2000, and T3E. In almost every case, a significant reduction in remapping time is consistently achieved when the adapted mesh is load balanced by performing data movement prior to refinement. This is because the mesh grows in size only after the data has been redistributed. The remapping times also usually decrease as the number of processors is increased because more processors are available to share the increase in the total volume of data movement. The remapping times when data is moved before mesh refinement are reproduced for the REAL_2R case in Table 8 since the exact values are difficult to read off the log-scale.

A peculiarity of these results is the behavior of the T3E when $P \geq 64$. When using up to 32 processors, the T3E closely follows the redistribution cost model given in § sec:costmodel; however, for 64 and 128 processors, the remapping overhead begins to increase even though the `MaxSR` metric continues to decrease. The runtime difference when data is remapped before and after refinement is dramatically diminished; in fact, all the remapping times begin to converge to a single value! This indicates that the remapping time is no longer affected only by the volume of data redistributed but also by the interprocessor communication pattern. One way of potentially improving these results is to take advantage of the T3E's ability to efficiently perform one-sided communication.

Another surprising result is the dramatic reduction in remapping times when using 32 processors on the Origin2000. This is probably because network contention with other jobs is essentially removed when using the entire machine. When using up to 16 processors, the remapping times on the SP2 and the Origin2000 are comparable, while the T3E is about twice as fast. Recall that the remapping phase within PLUM consists of both communication (to physically move data around) and computation (to rebuild the internal and shared

data structures). Since the results in Table 8 indicate that computation is faster on the Origin2000, it is reasonable to infer that bulk communication is faster on the SP2. These results generally demonstrate that our methodology within PLUM is effective in significantly reducing the data remapping time and improving the parallel performance of mesh refinement.

Table 8 also presents the ParMETIS partitioning times for REAL_2R on all three systems; the results for REAL_1R and REAL_3R are almost identical because the time to repartition mostly depends on the initial problem size. There is, however, some dependence on the number of processors used. When there are too few processors, repartitioning takes more time because each processor has a bigger share of the total work. When there are too many processors, an increase in the communication cost slows down the repartitioner. Table 8 demonstrates that ParMETIS is fast enough to be effectively used within our framework, and that PLUM can be successfully ported to different platforms without any code modifications.

## 5 Conclusions

Dynamic mesh adaptation on unstructured grids is a powerful tool for solving problems that require local grid modifications to efficiently resolve physical features of interest. For such problems, the coarsening/refinement step must be performed frequently, so its efficiency must be comparable to that of the numerical solver. Furthermore, with the ubiquity of parallel computing, it is imperative to have efficient parallel implementations of adaptive unstructured-grid algorithms. Unfortunately, parallel local mesh adaptation requires dynamic load balancing. In this paper, we described the parallel implementation of the 3D_TAG unstructured mesh adaptation algorithm and verified the effectiveness of our PLUM load balancer for a helicopter rotor blade acoustics problem.

Six refinement and two coarsening cases were presented with varying fractions of a realistic-sized domain being targeted for refinement. We demonstrated excellent parallel performance when repartitioning and remapping the mesh in a load balanced fashion after edges were targeted for refinement but before performing the actual subdivision. We presented three generic metrics to model the remapping cost on most multiprocessor systems. Optimal solutions for these metrics, as well as a heuristic approach were implemented. It was shown that our heuristic algorithm quickly finds a solution that satisfies all three metrics. Additionally, we showed that the data redistribution overhead can be significantly reduced by applying our heuristic processor reassignment algorithm to the default mapping given by the global partitioner. Portability was demonstrated by presenting results on the three vastly different archi-

tectures of the SP2, Origin2000, and T3E, without the need for any code modifications. Overall, the results showed that our parallel mesh adaptation and dynamic load balancing strategies will remain viable on large numbers of processors.

## References


[1] M.J. Berger and J.S. Saltzman, AMR on the CM-2, *Appl. Numer. Math.* **14** (1994) 239–253.

[2] K.S. Bey, J.T. Oden, and A. Patra, A parallel hp-adaptive discontinuous Galerkin method for hyperbolic conservation laws, *Appl. Numer. Math.* **20** (1996) 321–336.

[3] K. Bhat, *An $O(n^{2.5}\log_2 n)$ time algorithm for the bottleneck assignment problems* (AT&T Bell Laboratories, Murray Hill, NJ, 1984).

[4] R. Biswas and L. Dagum, Parallel implementation of an adaptive scheme for 3d unstructured grids on a shared-memory multiprocessor, in: A. Ecer, J. Periaux, N. Satofuka, and S. Taylor, eds., *Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers* (Elsevier, Amsterdam, The Netherlands, 1996) 489–496.

[5] R. Biswas, K.D. Devine, and J.E. Flaherty, Parallel, adaptive finite element methods for conservation laws, *Appl. Numer. Math.* **14** (1994) 255–283.

[6] R. Biswas and L. Oliker, Experiments with repartitioning and load balancing adaptive meshes, Numerical Aerospace Simulation Branch Tech. Rep. NAS-97-021, NASA Ames Research Center, Moffett Field, CA, 1997.

[7] R. Biswas and R.C. Strawn, A new procedure for dynamic adaption of three-dimensional unstructured grids, *Appl. Numer. Math.* **13** (1994) 437–452.

[8] R. Biswas and R.C. Strawn, Tetrahedral and hexahedral mesh adaptation for CFD problems, *Appl. Numer. Math.* **26** (1998) 135–151.

[9] J.G. Castanos and J.E. Savage, The dynamic adaptation of parallel mesh-based computation, *Proceedings 8th SIAM Conference on Parallel Processing for Scientific Computing* (SIAM, Minneapolis, MN, 1997).

[10] N. Chrisochoides, Multithreaded model for the dynamic load-balancing of parallel adaptive PDE computations, *Appl. Numer. Math.* **20** (1996) 349–365.

[11] H.L. de Cougny, K.D. Devine, J.E. Flaherty, R.M. Loy, C. Ozturan, and M.S. Shephard, Load balancing for the parallel adaptive solution of partial differential equations, *Appl. Numer. Math.* **16** (1994) 157–182.

[12] P. Diniz, S. Plimpton, B. Hendrickson, and R. Leland, Parallel algorithms for dynamically partitioning unstructured grids. *Proceedings 7th SIAM Conference on Parallel Processing for Scientific Computing* (SIAM, San Francisco, CA, 1995) 615–620.

[13] M.L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM* **34** (1987) 596–615.

[14] H.N. Gabow and R.E. Tarjan, Algorithms for two bottleneck optimization problems, *J. Alg.* **9** (1988) 411–417.

[15] H.N. Gabow and R.E. Tarjan, Faster scaling algorithms for network problems, *SIAM J. Comput.* **18** (1989) 1013–1036.

[16] J. Galtier, Automatic partitioning techniques for solving partial differential equations on irregular adaptive meshes, *Proceedings 10th ACM International Conference on Supercomputing* (ACM, Philadelphia, PA, 1996) 157–164.

[17] M.T. Jones and P.E. Plassmann, Parallel algorithms for adaptive mesh refinement, *SIAM J. Sci. Comput.* **18** (1997) 686–708.

[18] Y. Kallinderis and A. Vidwans, Generic parallel adaptive-grid Navier-Stokes algorithm, *AIAA J.* **32** (1994) 54–61.

[19] G. Karypis and V. Kumar, Parallel multilevel k-way partitioning scheme for irregular graphs, Department of Computer Science Tech. Rep. 96-036, University of Minnesota, Minneapolis, MN, 1996.

[20] G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM J. Sci. Comput.* **20** (1998) 359–392.

[21] T. Minyard and Y. Kallinderis, A parallel Navier-Stokes method and grid adapter with hybrid prismatic/tetrahedral grids, *Proceedings 33rd AIAA Aerospace Sciences Meeting* (AIAA, Reno, NV, 1995) Paper 95-0222.

[22] T. Minyard, Y. Kallinderis, and K. Schulz, Parallel load balancing for dynamic execution environments, *Proceedings 34th AIAA Aerospace Sciences Meeting* (AIAA, Reno, NV, 1996) Paper 96-0295.

[23] L. Oliker and R. Biswas, PLUM: Parallel load balancing for adaptive unstructured meshes, *J. Parallel Distrib. Comput.* **52** (1998) 150–177.

[24] T.W. Purcell, CFD and transonic helicopter sound, *14th European Rotorcraft Forum*. Milan, Italy, 1988, Paper 2.

[25] J.J. Quirk, A parallel adaptive grid algorithm for computational shock hydrodynamics, *Appl. Numer. Math.* **20** (1996) 427–453.

[26] K. Schloegel, G. Karypis, V. Kumar, R. Biswas, and L. Oliker, A performance study of diffusive vs. remapped load-balancing schemes, *Proc. 11th International Conference on Parallel and Distributed Computing Systems*. ISCA, Chicago, IL, 1998, pp. 59–66.

[27] P.M. Selwood, N.A. Verhoeven, J.M. Nash, M. Berzins, N.P. Weatherill, P.M. Dew, and K. Morgan, Parallel mesh generation and adaptivity: partitioning and analysis, in: P. Schiano, A. Ecer, J. Periaux, and N. Satofuka, eds., *Parallel Computational Fluid Dynamics: Algorithms and Results Using Advanced Computers* (Elsevier, Amsterdam, The Netherlands, 1997) 166–173.

[28] M.S. Shephard, J.E. Flaherty, H.L. de Cougny, C. Ozturan, C.L. Bottasso, and M.W. Beall, Parallel automated adaptive procedures for unstructured meshes, *Parallel Computing in CFD* AGARD-R-807 (1995) 6.1–6.49.

[29] H.D. Simon, A. Sohn, and R. Biswas, HARP: A dynamic spectral partitioner, *J. Parallel Distrib. Comput.* **50** (1998) 83–103.

[30] R.C. Strawn, R. Biswas, and M. Garceau, Unstructured adaptive mesh computations of rotorcraft high-speed impulsive noise, *J. Aircraft* **32** (1995) 754–760.

[31] L. Valiant, A bridging model for parallel computation, *Comm. ACM* **33** (1990) 103–111.

[32] R. Van Driessche and D. Roose, Load balancing computational fluid dynamics calculations on unstructured grids, *Parallel Computing in CFD* AGARD-R-807 (1995) 2.1–2.26.

[33] A. Vidwans, Y. Kallinderis, and V. Venkatakrishnan, Parallel dynamic load-balancing algorithm for three-dimensional adaptive unstructured grids, *AIAA J.* **32** (1994) 497–505.

[34] C. Walshaw, M. Cross, and M.G. Everett, Parallel dynamic graph partitioning for adaptive unstructured meshes, *J. Parallel Distrib. Comput.* **47** (1997) 102–108.