Proceedings of Supercomputing '96, Pittsburgh, PA, November 17-22, 1996.

Global Load Balancing with Parallel Mesh Adaption on Distributed-Memory Systems

Rupak Biswas
Research Institute for Advanced Computer Science
Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035
rbiswas@nas.nasa.gov

Leonid Oliker
Research Institute for Advanced Computer Science
Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035
oliker@riacs.edu

Andrew Sohn
Dept. of Computer and Information Science
New Jersey Institute of Technology, Newark, NJ 07102
sohn@cis.njit.edu

Abstract:

Dynamic mesh adaption on unstructured grids is a powerful tool for efficiently computing unsteady problems to resolve solution features of interest. Unfortunately, this causes load imbalance among processors on a parallel machine. This paper describes the parallel implementation of a tetrahedral mesh adaption scheme and a new global load balancing method. A heuristic remapping algorithm is presented that assigns partitions to processors such that the redistribution cost is minimized. Results indicate that the parallel performance of the mesh adaption code depends on the nature of the adaption region and show a 35.5X speedup on 64 processors of an SP2 when 35% of the mesh is randomly adapted. For large-scale scientific computations, our load balancing strategy gives almost a sixfold reduction in solver execution times over non-balanced loads. Furthermore, our heuristic remapper yields processor assignments that are less than 3% off the optimal solutions but requires only 1% of the computational time.

Keywords:

Load balancing, mesh adaption, unstructured grids, dual graph, processor reassignment, distributed-memory machines

Introduction

Dynamic mesh adaption on unstructured grids is a powerful tool for computing unsteady three-dimensional problems that require grid modifications to efficiently resolve solution features. By locally refining and coarsening the mesh to capture flowfield 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.

Unfortunately, the adaptive solution of unsteady problems causes load imbalance among processors on a parallel machine. This is because the computational intensity is not only time dependent, but also varies spatially over the problem domain. Dynamically balancing the computational load is, however, very difficult. It requires reliable measurements of processor workloads and the amount of data movement, as well as the minimization of interprocessor communication. Various methods on dynamic load balancing have been reported to date [3,4,6,7,9,10]; however, most of them lack a global view of loads across processors. A systematic way of measuring and balancing processor loads is needed for a method to be applicable to a variety of realistic applications.

Figure 1 depicts our framework for parallel adaptive flow computation. It consists of a flow solver and mesh adaptor, with a partitioner and mapper that redistributes the computational mesh when necessary. The mesh is first partitioned and mapped among the available processors. The flow solver then runs for several iterations, updating solution variables that are typically stored at the vertices of the mesh. Once an acceptable solution is obtained, the mesh adaption procedure is invoked. It targets edges for refinement or coarsening based on an error indicator computed from the flow solution. The old mesh is then locally adapted, generating a new computational mesh. 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 flow solver. Otherwise, a repartitioning procedure is invoked to divide the new mesh 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 flow calculation continues on the old partitions.

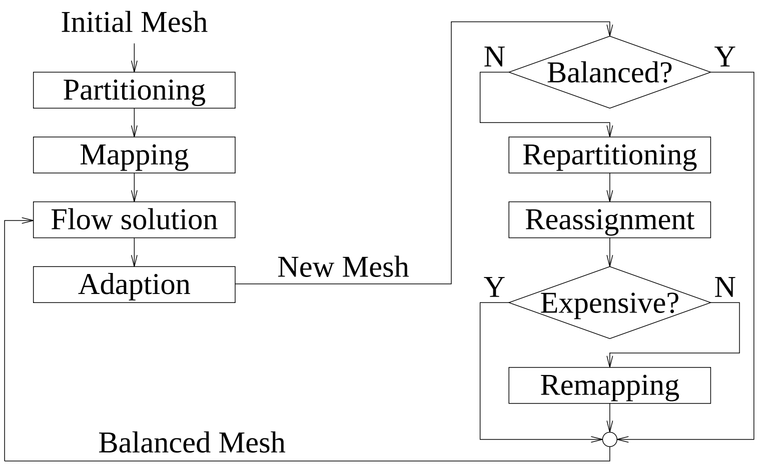

Figure 1: Overview of our framework for parallel adaptive flow computation.

Notice from the framework in Fig. 1 that the computational load is balanced and the runtime communication reduced only for the flow solver but not for the mesh adaptor. This is acceptable since the flow solver is usually several times more expensive. It is also obvious from Fig. 1 that mesh adaption, repartitioning, processor assignment, and remapping are critical components of the framework and must be accomplished rapidly and efficiently so as not to cause a significant overhead to the flow computation.

For parallel adaptive flow computations, the initial grid must first be partitioned among the available processors. A good partitioner should minimize the total execution time which is a function of load balance (computational time) and the interprocessor communication time. It is also important for our framework that the partitioning phase be performed rapidly. There are several excellent heuristic algorithms for solving the NP-hard graph partitioning problem [15]. Since mesh partitioning is not being addressed in this paper, we will assume that reasonable partitions for our test meshes are available, and address this issue in future work. For the record, we used the multilevel spectral Lanczos partitioning algorithm with local Kernighan-Lin refinement from the Chaco software package [8].

This paper briefly describes an efficient parallel implementation of a dynamic mesh adaption code which has shown good sequential performance on the C90 when coupled with a variety of unstructured flow solvers to solve realistic problems in helicopter and fixed-wing aerodynamics [1,2,5,14]. The parallel version consists of an additional 3,000 lines of C++ code with MPI, allowing portability to any system supporting these languages. This code is a wrapper around the original mesh adaption program written in C, and requires almost no changes to the serial version. Only a few lines were added to link it with the parallel constructs. An object-oriented approach allowed this to be performed in a clean and efficient manner. Complete details are given in [11].

This paper also describes a new method that has been developed to dynamically balance the processor workloads with a global view. The load-balancing procedure uses a dual graph representation of the computational mesh in order to keep the complexity and connectivity constant during the course of an adaptive computation. It uses heuristic but accurate metrics to estimate the computational gain and the redistribution cost of having a balanced workload after each mesh adaption. Even though mesh repartitioning is an inherent component of our global load balancing scheme, it is not addressed in this paper but will be the focus in subsequent work. A concise description of the new inertial spectral mesh repartitioning method applied to small model meshes is given in [13].

Tetrahedral Mesh Adaption

We give a brief description of the tetrahedral mesh adaption scheme; complete details are given in [1]. The code, called 3D_TAG, has its data structures based on edges that connect the vertices of a tetrahedral mesh. This means that the elements and boundary faces are defined by their edges rather than by their vertices. These edge-based data structures make the mesh adaption procedure capable of performing anisotropic refinement and coarsening that results in a more efficient distribution of grid points.

At each mesh adaption step, tetrahedral elements are targeted for coarsening, refinement, or no change by computing an error indicator for each edge. Edges whose error values exceed a specified upper threshold are targeted for subdivision. Similarly, edges whose error values lie below another lower threshold are targeted for removal. Only three subdivision types are allowed for each 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 vector is linearly interpolated at the mid-point from the two points that constitute the original edge.

Mesh refinement is performed by first setting a bit flag to one for each edge that is targeted

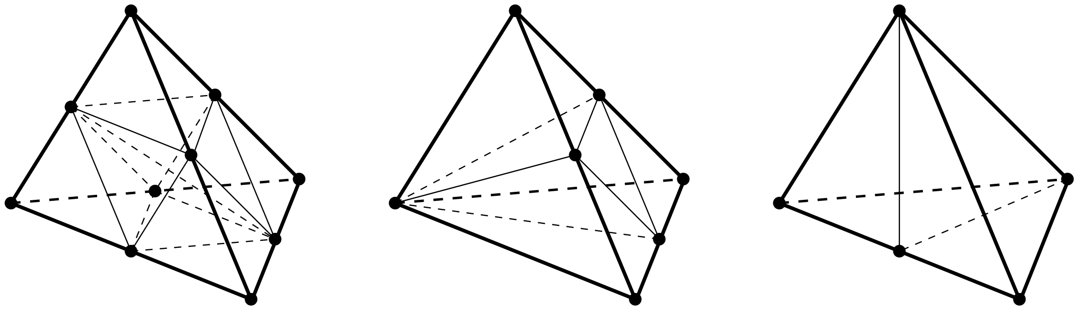

Figure 2: Three types of subdivision are permitted for a tetrahedral element.

for subdivision. The edge markings for each element are then combined to form a 6-bit binary pattern. Elements are continuously upgraded to valid patterns corresponding to the three allowed subdivision types shown in Fig. 2 until none of the patterns show any change. Once this edge-marking is completed, each element is independently subdivided based on its binary pattern.

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 element is reinstated. The 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 mesh refinement procedure is then invoked to generate a valid mesh. Note that edges cannot be coarsened beyond the initial mesh.

Pertinent information is maintained for the vertices, elements, edges, and external boundary faces of the mesh. In addition, each vertex has a list of all the edges that are incident upon it. Similarly, each edge has a list of all the elements that share it. These lists eliminate extensive searches and are crucial to the efficiency of the overall adaption scheme.

Parallel Implementation

The distributed-memory implementation of the 3D_TAG mesh adaption code consists of three phases: initialization, execution, and finalization. The initialization and finalization steps are executed only once for each problem outside the main solution$\leftrightarrow$adaption cycle shown in Fig. 1. The execution step runs a local copy of 3D_TAG on each processor. Good parallel performance is therefore critical during this phase since it is executed several times during a flow computation.

The initialization phase takes as input the global initial grid and the corresponding partitioning information that places each tetrahedral element in exactly one partition. It then distributes the global data across the processors, defining a local number for each mesh object, and creating the mapping for objects that are shared by multiple processors. Shared vertices and edges are identified by searching for elements that lie on partition boundaries. A bit flag is set to distinguish between shared and internal objects. A list of shared processors (SPL) is also generated for each shared object. The maximum additional storage that is required for the parallel code 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.

The execution phase 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. The first step is to target edges for refinement or coarsening based on an error indicator for each edge that is computed from the flow solution. This process results in a symmetrical marking of all shared edges across partitions because shared edges have the same flow and geometry 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 for refinement 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. The process may continue for several iterations, and edge markings could propagate back and forth across partitions.

Figure 3 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 for refinement during this phase. Information about the shared edge 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.

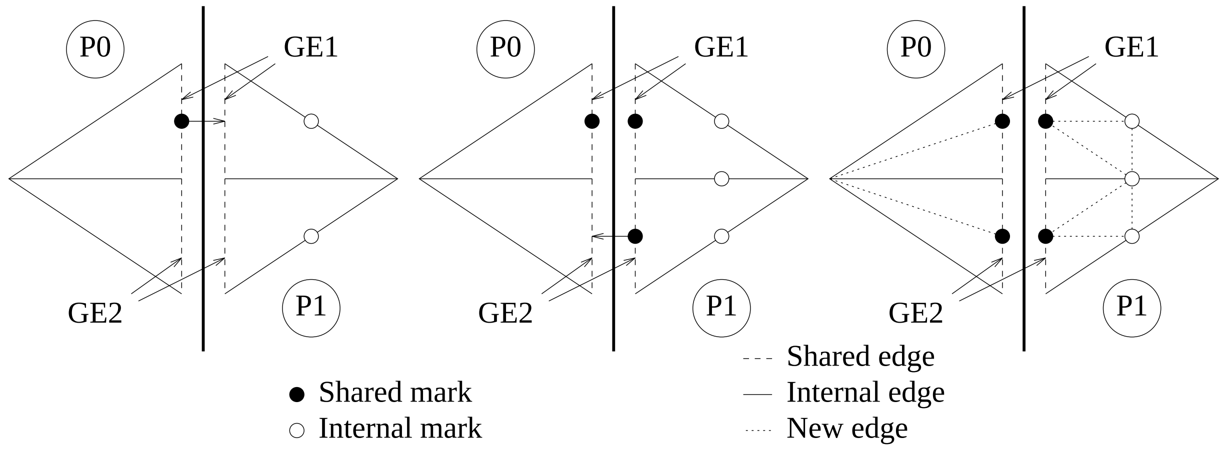

Figure 3: A two-dimensional example showing communication due to the propagation of edge markings.

Once all edge markings are complete, each processor executes the mesh adaption 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 faces of the tetrahedral elements. An alternate solution would be to incorporate faces as an additional object into the data structures, and maintaining it through the adaption. However, this does not compare favorably in terms of memory or CPU time to a single communication at the end of the refinement procedure.

Figure 4 shows the top view of a tetrahedron in processor P0 that shares two faces with P1. In P0, the intersection of the SPLs for the end-points of all the three new edges LE1, LE2, and LE3 yields P1. However, when P0 communicates this information to P1, P1 will only have local copies corresponding to LE1 and LE2. Thus, P0 will classify LE1 and LE2 as shared edges but LE3 as an internal edge.

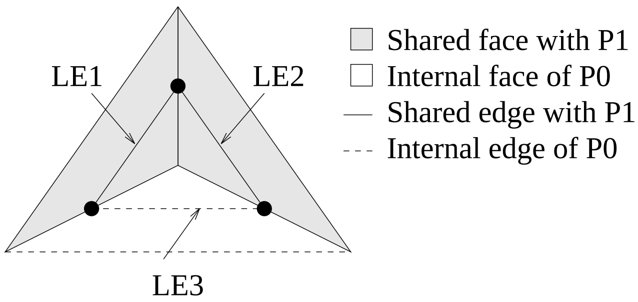

Figure 4: Example showing how a new edge that lies across a face is classified as shared or internal.

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 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.

It is sometimes necessary to create a single global mesh after one or more adaption steps. Some post processing tasks, such as visualization, need to processes the whole grid simultaneously. Storing a snapshot of a grid for future restarts could also require a global view. The finalization phase accomplishes this task by connecting individual subgrids into one global mesh. Each local object is first assigned a unique global number. Details of how global numbers are assigned are given in [11]. All processors then update their local data structures accordingly. Finally, a gather operation is performed by a host processor to concatenate the local data structures into a global mesh. The host can then interface the mesh directly to the appropriate post-processing module without having to perform any serial computation.

Dual Graph Representation

The dual graph representation of the initial computational mesh is one of the key features of this work. Parallel implementation of adaptive flow solvers requires a partitioning of the computational mesh such that each element belongs to an unique partition. Communication is required across faces that are shared by adjacent tetrahedral elements residing on different processors. Hence for the purposes of partitioning, we consider the dual of the original computational mesh. The tetrahedral elements of the computational mesh are the vertices of the dual graph. An edge exists between two dual graph vertices if the corresponding elements share a face. A graph partitioning of the dual thus yields an assignment of tetrahedra to processors.

Each dual graph vertex has two weights associated with it. The computational weight, $w_{\rm comp}$, indicates the workload for the corresponding element. The remapping weight, $w_{\rm remap}$, indicates the cost of moving the element from one processor to another. The weight $w_{\rm comp}$ is set to the number of leaf elements in the refinement tree because only those elements that have no children participate in the flow computation. The weight $w_{\rm remap}$, however, is set to the total number of elements in the refinement tree because all descendants of the root element must move with it from one partition to another if so required. The connectivity and $w_{\rm comp}$ determine how dual graph vertices should be grouped to form partitions that minimize the disparity in the partition weights. The $w_{\rm remap}$ determines how partitions should be assigned to processors such that the cost of data movement is minimized.

Every edge in the dual graph also has a weight that models the runtime communication. This information is used by the mesh partitioner along with the computational weights of the dual graph vertices to balance the processor workloads and minimize the runtime communication. The edge weights are uniform for the test cases in this paper.

The most significant advantage of using the dual of the initial computational mesh is that its complexity and connectivity remains unchanged during the course of an adaptive computation. The partitioning and load-balancing times therefore depend only on the initial problem size and the number of partitions. New grids obtained by adaption are translated to the two weights, $w_{\rm comp}$ and $w_{\rm remap}$, for every element in the initial mesh.

One minor disadvantage of using the dual grid is when the initial computational mesh is either too large or too small. For extremely large initial meshes, the partitioning time will be excessive. This problem can be circumvented by agglomerating groups of elements into larger superelements. For very small meshes, the quality of the partitions will be bad. One can then allow the initial mesh to be adapted one or more times before using the dual graph for all future adaptions.

Preliminary Evaluation

The preliminary evaluation step rapidly determines if the dual graph with a new set of $w_{\rm comp}$ should be repartitioned. If projecting the new values on the current partitions indicates that they are adequately load balanced, there is no need to repartition the mesh. In that case, the flow computation continues uninterrupted on the current partitions.

A proper metric is required to measure the load imbalance. If $W_{\max}$ is the sum of the w_{comp} on the most heavily-loaded processor, and W_{avg} is the average load across all processors, the average idle time for each processor is $(W_{\max} - W_{\text{avg}})$. This is an exact measure of the load imbalance. The mesh is repartitioned if the imbalance factor $W_{\max}/W_{\text{avg}}$ is greater than a specified threshold.

Similarity Matrix Construction

If the preliminary evaluation phase determines that the dual graph with the new w_{comp} is not adequately load balanced, the mesh is repartitioned to balance the processor workloads. Any mesh partitioning algorithm can be used here, as long as it quickly delivers partitions that are reasonably balanced.

Once new partitions are obtained, they must be mapped to the processors such that the redistribution cost is minimized. We assume that the redistribution cost is proportional to the volume of data moved. In the simplest case, the number of new partitions is equal to the number of processors. In our general framework, however, it is possible to have the number of partitions be an integer multiple F of the number of processors, and then map more than one partition to a processor. The rationale behind allowing multiple partitions per processor is that performing data mapping at a finer granularity results in a smaller volume of data movement at the expense of processor reassignment time. However, the simpler scheme of setting F to unity suffices for most practical applications.

The first step toward processor reassignment is to compute a similarity measure S that indicates how the remapping weights w_{remap} of the new partitions are distributed over the processors. It is represented as a matrix of P rows and $P \times F$ columns, where P is the number of processors. Each entry S_{ij} is the sum of the w_{remap} of all the dual graph vertices that are common between processor i and new partition j. Therefore, the sum of the entries in row i is the total remapping weight of all the dual graph vertices currently residing on processor i. A similarity matrix for a remapping of eight partitions on four processors is shown in Fig. 5. Only the non-zero entries are shown.

New Partitions

Old Processors	0	1	2	3	4	5	6	7
0		1020		120				
1			500		443	372		
2	129	130		229			43	446
3	13	410	281				198	

Figure 5: An example of a similarity matrix S for $P = 4$ and $F = 2$. The F largest weights for each processor are shaded.

Processor Reassignment

A new partition j with the largest value of S_{ij} is called the dominant partition for processor

i. The overhead for data movement from processor i can be minimized by reassigning it to its dominant partition. To minimize the total data movement for all processors when $F = 1$, each processor i must be assigned to an unique partition j_i so that the objective function $\mathcal{F} = \sum_{i=1}^{P} S_{ij_i}$ is maximized subject to the constraint $j_i \neq j_m, \forall i \neq m$. In general, each processor i is assigned to exactly F unique partitions $j_{i(1)}, j_{i(2)}, \ldots, j_{i(F)}$ so that the objective function

$$\mathcal{F} = \sum_{i=1}^{P} \sum_{k=1}^{F} S_{ij_{i(k)}}$$

is maximized subject to

$$j_{i(p)} \neq j_{m(q)}, \quad \forall i(p) \neq m(q); \quad p = 1, 2, \ldots, F; \quad q = 1, 2, \ldots, F.$$

Both an optimal and a heuristic greedy algorithm have been implemented for solving this problem. When $F = 1$, the problem trivially reduces to solving a maximally weighted bipartite graph, with P processors and P partitions in each set. An edge of weight S_{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 the maximally weighted bipartite graph problem 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 pseudocode for our heuristic algorithm is as follows:

```
                                                         /* initialization */
for (j=0; j<#partitions; j++) partition_map[j] = unassigned;
for (i=0; i<#processors; i++) total_unmapped[i] = #partitions / #processors;

while (there exists an unassigned partition) {
  for (i=0; i<#processors; i++)                                    /* mark */
    for (k=0; k<total_unmapped[i]; k++)
      mark largest entry S[i][j] such that partition_map[j] == unassigned;

  foreach (j such that partition_map[j] == unassigned)              /* map */
    if (there exists at least one marked entry in column j) {
      find the largest marked entry S[i][j];
      total_unmapped[i]--;
      partition_map[j] = assigned; }
}
```

The heuristic algorithm consists of an initialization step, followed by repeated iterations of the mark and map steps. Initially, all partitions are considered unassigned and each processor has a counter set to F that indicates the remaining number of partitions it needs to be assigned. In the marking phase, each processor that has less then F assigned partitions marks the necessary number of largest entries in S from the set of unassigned partitions. The mapping phase examines all the available partitions j that have at least one marked entry. The largest entry S_{ij} is chosen and partition j is assigned to processor i. This results in partition j becoming unavailable and processor i requiring one less partition assignment. The mark and map steps are repeated until all partitions are assigned. Applying our heuristic algorithm to the similarity matrix in Fig. 5 generates the new processor assignment shown in Fig. 6(a). The optimal assignment in shown in Fig. 6(b). The value of the objective function $\mathcal{F}$ is 2849 for the heuristic algorithm but 2989 for the optimal solution.

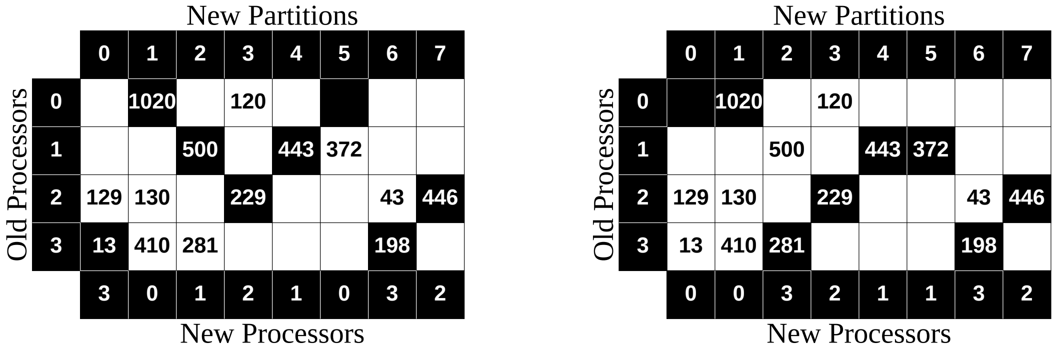

Figure 6: The similarity matrix S after processor assignment using the (a) heuristic and (b) optimal algorithms.

We claim that our heuristic algorithm can never give a processor assignment that results in a data movement cost that is more than twice the optimal cost. Given a similarity matrix S, the heuristic algorithm initially assigns processor i to partition j such that S_{ij} has the largest value L in row i and column j. Partition j is then removed from the available list. Assume that the optimal algorithm maps processor i to partition k and processor l to partition j. However, the values of S_{ik} and S_{lj} are bounded by L. If, in the worst case, all partition-to-processor assignments are chosen incorrectly, the heuristic algorithm gives a solution that is twice as expensive as the optimal solution.

Cost Calculation

The computational gain due to repartitioning is proportional to the decrease in the load imbalance achieved by running the adapted mesh on the new partitions rather than on the old partitions. Recall from Sec. 5 that the average load imbalance for each processor is given by $(W_{\max} - W_{\text{avg}})$. The decrease in load imbalance due to the new partitioning is therefore $(W_{\max}^{\text{old}} - W_{\max}^{\text{new}})$, where $W_{\max}^{\text{old}}$ and $W_{\max}^{\text{new}}$ are the sum of the w_{comp} on the most heavily-loaded processor for the old and new partitionings, respectively. If it requires T_{iter} secs to run one iteration of the flow solver on one element of the original mesh, and if it is expected that the next mesh adaption will occur after N_{adapt} solver iterations, the total computational gain for the new partitioning is $T_{\text{iter}} N_{\text{adapt}} (W_{\max}^{\text{old}} - W_{\max}^{\text{new}})$.

The redistribution cost is calculated from the similarity matrix obtained after processor reassignment. Two machine-dependent parameters are used to calculate the actual cost: the remote-memory latency time T_{lat} and the message setup time T_{setup}. T_{lat} is the time required for memory-to-memory copying of a word, and applies to every dual grid vertex that is moved. T_{setup} is the time required to prepare message headers, load the message buffer, and so on, and applies to each set of elements that is moved from one processor to another. If the flow solver and mesh adaptor require M words of storage per element, and if $C = (\sum \sum S_{ij} - \mathcal{F})$ and N are the total number of elements and sets of elements to be moved, respectively (cf. Fig. 7), the total communication overhead for mapping new partitions to processors is $CMT_{\text{lat}} + NT_{\text{setup}}$. Since the quantity CM is typically much larger than N for realistic problems, the second term can be neglected.

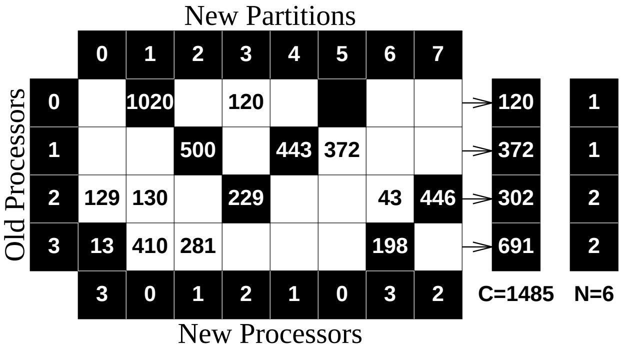

Figure 7: Calculating the total redistribution cost from the similarity matrix S. Note that two (not three) sets of elements are moved from old processor 2 because new partitions 0 and 6 are both mapped to new processor 3.

The new partitioning and mapping are accepted if the computational gain is larger than the redistribution cost:

$$T_{\text{iter}} N_{\text{adapt}} (W_{\text{max}}^{\text{old}} - W_{\text{max}}^{\text{new}}) > C M T_{\text{lat}} + N T_{\text{setup}}.$$

The numerical simulation is then interrupted to properly redistribute all the data.

Remapping

The remapping phase is responsible for physically moving the data when it is reassigned to a different processor. When an element is moved to a different processor, two kinds of overhead are incurred: communication and computation. The communication overhead includes the cost of packing and unpacking the send and receive buffers, as well as the message setup time and the remote-memory latency time. The computation cost is the time necessary to rebuild the internal and shared data structures in a consistent manner.

The remapping procedure used in the experiments reported in this paper is not fully operational; however, it does predict the cost with reasonable accuracy. Based on the processor reassignments, all appropriate mesh objects are sent to their new host processor, accurately modeling the communication phase. Note that the relationship between the number of elements moved and the total data volume is not exactly linear. This is due to the movement of the shared data structures whose size is a function of the locations of the old and new partition boundaries. The shared information accounts for a small percentage of the data volume, and is the cause of the slight perturbations.

The computation phase in not yet complete, and data structures are only partially restored after the data movement. Since communication accounts for the majority of the remapping overhead, we expect the simulated remapping time to be within 10% of the fully functional procedure. The implementation of this phase will be completed shortly.

Results

The parallel 3D_TAG and global load balancing procedures have been implemented on an IBM SP2 distributed-memory multiprocessor. Both codes are written in C and C++, with the parallel activities in MPI for portability. Note that no SP2-specific optimizations were used to obtain the performance results reported in this section.

The computational mesh is the one used to simulate the acoustics experiment of Purcell [12] where a 1/7th scale model of a UH-1H helicopter rotor blade was tested over a range of subsonic and transonic hover-tip Mach numbers. Numerical results and a detailed report of the simulation are given in [14].

Results are presented for one refinement and one coarsening step using three different edge-marking strategies. The first strategy, called `Local_1`, targeted 5% of the edges for refinement in a single spherical region of the mesh. The subsequent coarsening phase undid all of the refinement to restore the initial mesh. The second strategy, called `Local_2`, refined 35% of the edges in a single rectangular region of the mesh. Coarsening was performed within a rectangular subregion. The third strategy, called `Random`, consisted of randomly targeting edges for adaption such that the mesh sizes after both refinement and coarsening were approximately equal to those obtained in the `Local_2` case. These strategies represent significantly different scenarios. In general, real edge-marking patterns are expected to lie somewhere between `Local_1` and `Local_2`. Table 1 presents the progression of grid sizes through the two adaption steps for each marking strategy.

	`Local_1`		`Local_2`		`Random`	
	Elements	Edges	Elements	Edges	Elements	Edges
Initial Mesh	60,968	78,343	60,968	78,343	60,968	78,343
After Refinement	82,259	104,178	201,543	246,112	201,734	246,949
After Coarsening	60,968	78,343	100,241	125,651	100,537	126,448

Table 1: Progression of grid sizes through refinement and coarsening.

Figure 8 illustrates the parallel speedup of the refinement and coarsening phases of the 3D_TAG code for the three edge-marking strategies. As expected, `Random` gives the best speedup performance as the processor workloads are inherently balanced. Note that our load balancing scheme only balances the load for the flow solver after the mesh adaption step is completed. The refinement speedup results are the worst for the `Local_1` case because a compact spherical region of the mesh is adapted. All of the work is thus performed by only a handful of processors. The coarsening results are similar to those of the refinement step because of the algorithmic similarities of the two methods. However, performance improves significantly for the `Local_1` strategy. This is because the processor workloads are better balanced as coarsening undoes all of the previous refinement. Extensive performance analysis of the parallel 3D_TAG code is given in [11].

Figure 9 shows how the execution time is spent during the refinement and the subsequent load-balancing phases of the `Local_1` and `Local_2` strategies. The repartitioning times are not shown as it is not the focus of this paper. As mentioned in Sec. 9, the remapping time consists of communication and computation overheads. Note that the remapping time initially increases with the number of processors, but then gradually decreases. This is not entirely unexpected. Even though the total volume of data movement increases with the number of processors, there are actually more processors to share the work. This indicates that our global load balancing strategy will remain viable on large numbers of processors as

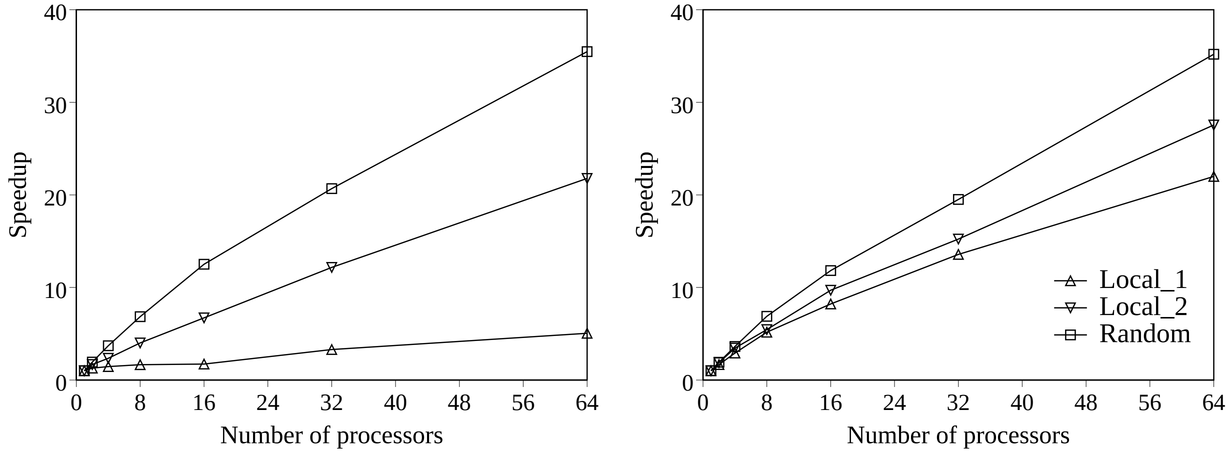

Figure 8: Speedup of the parallel mesh adaption code during the (a) refinement and (b) coarsening stages.

the remapping phase will not become a bottleneck. The speedup curves in Fig. 8 also show that the mesh adaption time decreases as more processors are used. Although the processor reassignment time increases with the number of processors used, it remains negligible compared to the adaption and remapping times even for 64 processors. The curves in Fig. 9 are for $F = 1$ using our heuristic processor reassignment algorithm. Similar results were obtained for the coarsening phase of the mesh adaption procedure.

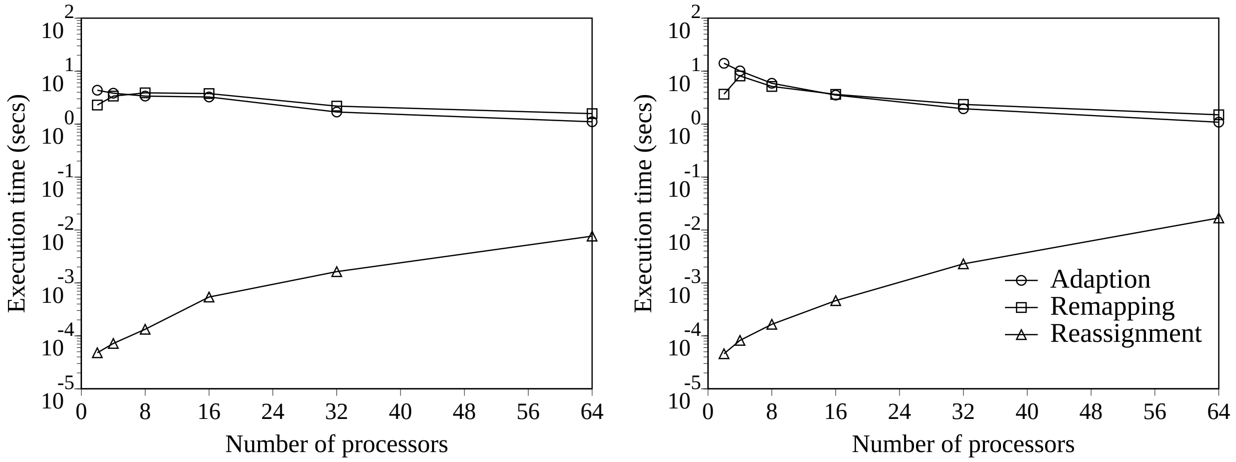

Figure 9: Anatomy of total execution times for the (a) `Local_1` and (b) `Local_2` refinement strategies.

Figure 10 compares the execution times and the amount of data movement for the `Local_2` strategy when using the optimal and heuristic processor assignment algorithms. Four sets of curves are shown in each plot for $F = 1, 2, 4$, and 8. The optimal method always requires almost two orders of magnitude more time than our heuristic method. The execution times also increase significantly as F is increased. This is because the size of the similarity matrix grows with F. However, the volume of data movement decreases with increasing F. This confirms our earlier claim that data movement can be reduced by mapping at a finer granularity. The relative reduction in data movement, however, is not very significant for our test cases. The results in Fig. 10 illustrate that our heuristic mapper is almost as good as the optimal algorithm while requiring significantly less time. Similar results were obtained for the `Local_1` and `Random` edge-marking strategies.

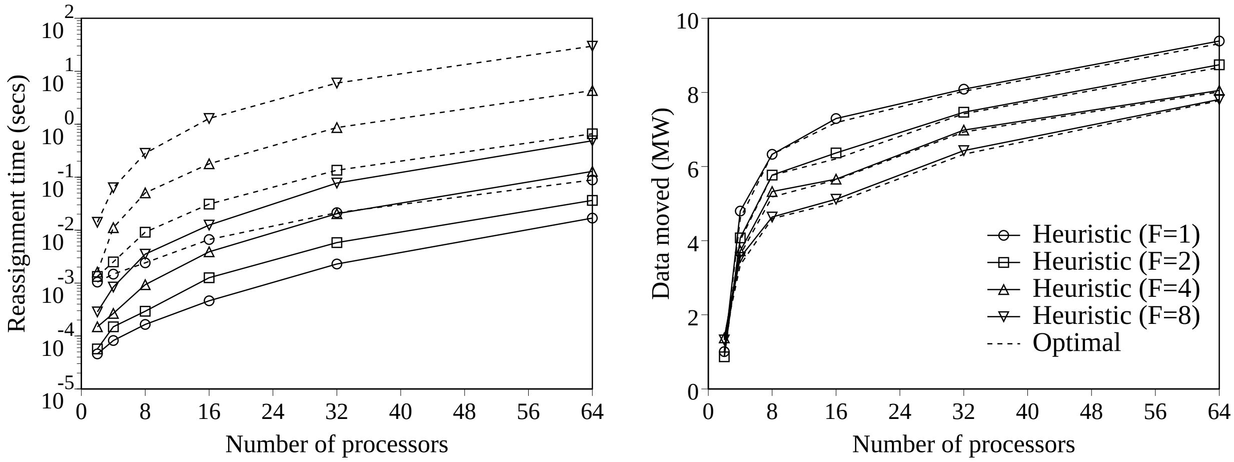

Figure 10: Comparison of the optimal and heuristic mappers in terms of (a) execution time and (b) volume of data movement for the `Local_2` refinement strategy.

Figure 11 shows the relationship between the remapping time and the number of tetrahedral elements that are moved from one processor to another. Individual data points on the curves are obtained by varying F. As shown earlier in Fig. 10(b), increasing F reduces the number of elements moved. Note, however, that the remapping time sometimes increases even when fewer elements are moved. This is due to the computational requirements of the remapper as described in Sec. 9. The plots demonstrate that for a given number of processors, there is a strong correlation between the number of elements moved and the remapping time. This confirms our two earlier claims. First, a good solution to the similarity matrix reduces the remapping times. Second, the total number of elements moved can be scaled by a factor to give a good approximation of the remapping time. This supports our evaluation model which predicts whether a balanced load is worth the expense of remapping.

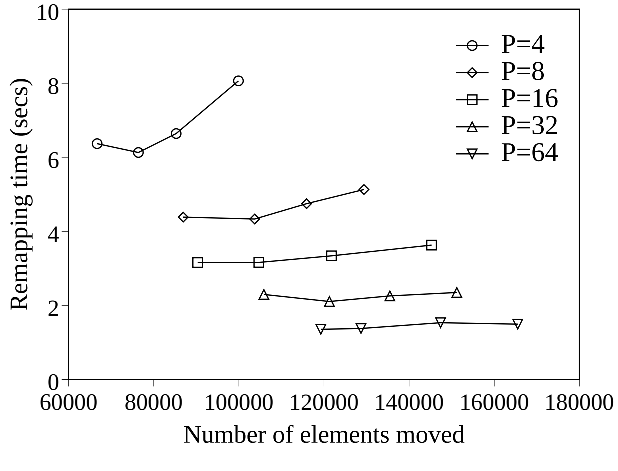

Figure 11: Variation of the remapping time with the number of elements moved for the `Local_2` refinement strategy.

Finally, Fig. 12 illustrates the impact of load balancing on the execution time of the flow solver. Note that the maximum possible improvement is not linear. It can be explained as follows. Suppose that there are P processors and that each processor has N elements assigned to it. In the worst case, all N elements on only one processor are isotropically refined (cf. Fig. 2) using 3D_TAG to generate $8N$ elements while none of the other elements are refined. If the adapted mesh is not load balanced, the flow solution would require time proportional to $8N$, the most heavily-loaded processor. However, if the mesh were balanced, each processor would have $\frac{8N+(P-1)N}{P}$ elements. Thus, load balancing would give an improvement of $\frac{8P}{P+7}$ over a non-balanced load.

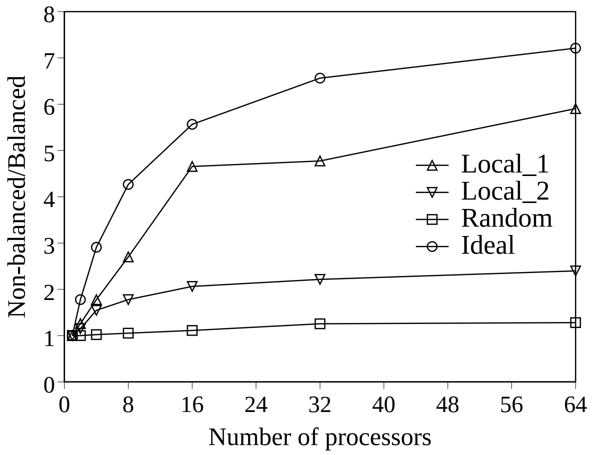

Figure 12: Comparison of flow solver execution times with and without load balancing.

Note from Fig. 12 that the `Random` case gives only a marginal improvement when the processor loads are balanced. This is expected because the computational work is already distributed uniformly among the processors after the mesh is adapted. `Local_1` shows the best improvement with load balancing because a small compact region of the mesh was refined that led to a severe imbalance among the processors. With 64 processors, the improvement is almost sixfold. It is important to realize that the results shown in Fig. 12 are for a single refinement step. With repeated adaption, the gains realized with load balancing may be even more significant.

Summary

Fast and efficient dynamic mesh adaption is an important feature of unstructured grids that makes them especially attractive for unsteady flows. However, mesh adaption on parallel computers can cause serious load imbalance among the processors. Dynamically balancing the processor loads at runtime is a complex task.

We have described a distributed-memory implementation of an edge-based adaption scheme that has shown good single-processor performance on the C90. The code is written in C and C++ using the MPI message-passing paradigm. Performance results on an SP2 show a 35.5X speedup on 64 processors when about 35% of a helicopter rotor mesh containing more than 60,000 tetrahedral elements and 78,000 edges is randomly adapted. The speedup is reduced to about 25.0X due to load imbalance when the same number of edges is refined in a single compact region of the mesh.

We have also described a new dynamic load balancing scheme that balances the processor workloads with a global view. The procedure uses a dual graph representation of the computational mesh to keep the complexity and connectivity constant during the course of an adaptive computation. Each time the computational mesh is adapted, the load balancer is invoked to determine if the new mesh warrants repartitioning. New partitions obtained by repartitioning are assigned to processors using a heuristic algorithm that strives to minimize the amount of data movement.

Results have demonstrated that the remapping time decreases with the number of processors, indicating that our global load balancing strategy will remain viable on massively-parallel systems. Although the processor reassignment time increases as more processors are used, it remains negligible compared to the adaption and remapping times. Our heuristic remapper has been shown to yield processor assignments that differ from optimal solutions by less than 3% but requires only 1% of the computational times. Finally, large-scale scientific computations on 64 processors of an SP2 show that load balancing gives almost a sixfold reduction in flow solver execution times over non-balanced loads. With multiple mesh adaptions, the gains realized with load balancing may be even more significant.

Acknowledgement

The authors would like to thank Rob Schreiber for observing that our processor remapping problem is equivalent to a weighted bipartite graph.

References

[1] R. Biswas and R. Strawn, **A New Procedure for Dynamic Adaption of Three-Dimensional Unstructured Grids**, Applied Numerical Mathematics, Vol. 13, pp. 437–452, 1994.

[2] R. Biswas, S. Thomas, and S. Cliff, **An Edge-Based Solution-Adaptive Method Applied to the AIRPLANE Code**, 34th AIAA Aerospace Sciences Meeting, Paper 96-0553, 1996.

[3] G. Cybenko, **Dynamic Load Balancing for Distributed-Memory Multiprocessors**, Journal of Parallel and Distributed Computing, Vol. 7, pp. 279–301, 1989.

[4] Y. Deng, R. McCoy, R. Marr, and R. Peierls, **An Unconventional Method for Load Balancing**, 7th SIAM Conference on Parallel Processing for Scientific Computing, pp. 605–610, 1995.

[5] E. Duque, R. Biswas, and R. Strawn, **A Solution Adaptive Structured/Unstructured Overset Grid Flow Solver with Applications to Helicopter Rotor Flows**, 13th AIAA Applied Aerodynamics Conference, Paper 95-1766, 1995.

[6] K. Devine, J. Flaherty, S. Wheat, and A. Maccabe, **A Massively Parallel Adaptive Finite Element Method with Dynamic Load Balancing**, Supercomputing, pp. 2–11, 1993.

[7] B. Ghosh and S. Muthukrishnan, **Dynamic Load Balancing in Parallel and Distributed Networks by Random Matchings**, 6th ACM Symposium on Parallel Algo-

rithms and Architectures, pp. 226–235, 1994.

[8] B. Hendrickson and R. Leland, **The Chaco user's guide — Version 2.0**, Sandia National Laboratories, Technical Report SAND94-2692, 1994.

[9] G. Horton, **A Multilevel Diffusion Method for Dynamic Load Balancing**, Parallel Computing, Vol. 19, pp. 209–229, 1993.

[10] R. Knecht and G. Kohring, **Dynamic Load Balancing for the Simulation of Granular Materials**, 9th ACM International Conference on Supercomputing, pp. 164–169, 1995.

[11] L. Oliker, R. Biswas, and R. Strawn, **Parallel Implementation of an Adaptive Scheme for 3D Unstructured Grids on the SP2**, 3rd International Workshop on Parallel Algorithms for Irregularly Structured Problems, to appear, 1996.

[12] T. Purcell, **CFD and Transonic Helicopter Sound**, 14th European Rotorcraft Forum, Paper 2, 1988.

[13] A. Sohn, R. Biswas, and H. Simon, **Impact of Load Balancing on Unstructured Adaptive Grid Computations for Distributed-Memory Multiprocessors**, 8th IEEE Symposium on Parallel and Distributed Processing, to appear, 1996.

[14] R. Strawn, R. Biswas, and M. Garceau, **Unstructured Adaptive Mesh Computations of Rotorcraft High-Speed Impulsive Noise**, Journal of Aircraft, Vol. 32, pp. 754–760, 1995.

[15] R. Van Driessche and D. Roose, **Load Balancing Computational Fluid Dynamics Calculations on Unstructured Grids**, AGARD, Report R-807, 1995.