Proceedings of the 3rd International Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'96), Santa Barbara, CA, August 19–21, 1996.

Parallel Implementation of an Adaptive Scheme for 3D Unstructured Grids on the SP2

Leonid Oliker[1], Rupak Biswas[1], and Roger C. Strawn[2]

[1] RIACS, NASA Ames Research Center, Moffett Field, CA 94035, USA
[2] US Army AFDD, NASA Ames Research Center, Moffett Field, CA 94035, USA

Abstract. Dynamic mesh adaption on unstructured grids is a powerful tool for computing unsteady flows that require local grid modifications to efficiently resolve solution features. For this work, we consider an edge-based adaption scheme that has shown good single-processor performance on the C90. We report on our experience parallelizing this code for the SP2. Results show a 47.0X speedup on 64 processors when 10% of the mesh is randomly refined. Performance deteriorates to 7.7X when the same number of edges are refined in a highly-localized region. This is because almost all the mesh adaption is confined to a single processor. However, this problem can be remedied by repartitioning the mesh immediately after targeting edges for refinement but before the actual adaption takes place. With this change, the speedup improves dramatically to 43.6X.

1 Introduction

Unstructured grids 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 adaption, 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 added 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 adaption procedures have proved to be robust, reliable, and efficient. By redistributing the available mesh

points 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.

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 easily programmed with regular data structures; so the development of efficient parallel adaptive algorithms for unstructured grids poses a serious challenge.

Figure 1 depicts our framework for parallel adaptive flow computation. 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. The flow solver then runs for several iterations, updating solution variables that are typically stored at the vertices of the mesh. If desired, local mesh adaption is then performed, 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 mesh repartitioning procedure is invoked to divide the new grid into subgrids. 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. 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.

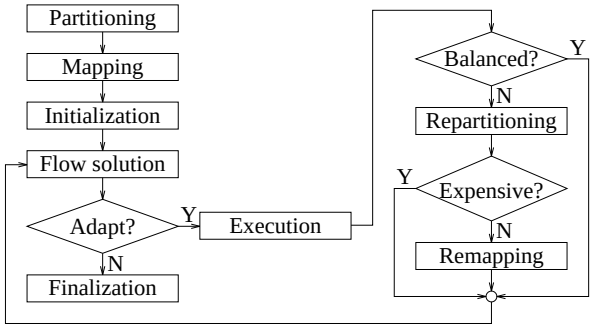


Fig. 1. Overview of our framework for parallel adaptive flow computation

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. However, parallel performance for the mesh adaption 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 adaption.

It is obvious from Fig. 1 that a quick mesh adaption procedure is a critical part of the framework. This paper presents an efficient parallel implementation of a dynamic mesh adaption code which has shown good sequential performance. The parallel version consists of 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 adaption program written in C, and requires almost no changes to the serial code. 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.

2 Mesh Adaption Procedure

We give a brief description of the tetrahedral mesh adaption scheme [1] that is used in this work to better explain the modifications that were made for the distributed-memory implementation. 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.

At each mesh adaption step, individual edges are marked for coarsening, refinement, or no change. 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.

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. Elements are continuously upgraded to valid patterns corresponding to the three allowed subdivision types until none of the

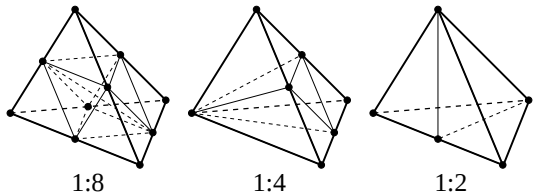


Fig. 2. Three types of subdivision are permitted for a tetrahedral element

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 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 refinement procedure is then invoked to generate a valid mesh.

Details of the data structures are given in [1]; however, a brief description of the salient features is necessary to understand the distributed-memory implementation of the mesh adaption code. For each vertex, a pointer to the first entry in the edge sublist is stored in `edges`. The edge sublist for a vertex contains pointers to all the edges that are incident upon it. Such sublists eliminate extensive searches and are crucial to the efficiency of the overall adaption scheme. For each edge, we store its two end-points in `vertex[2]`, the two boundary faces it defines in `bfac[2]`, and a pointer to the first entry in the element sublist in `elems`. The element sublist for an edge contains pointers to all the elements that share it. The tetrahedral elements have their six edges stored in `tedge[6]`, while for each boundary face, we store the three edges in `bedge[3]`.

3 Distributed-Memory Implementation

The parallel implementation of the 3D_TAG mesh adaption code 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 flow computation. Finally, a gather operation is performed in the finalization step to combine the local grids into one global mesh. Locally-numbered objects and corresponding pointers are reordered to represent one single consistent mesh.

In order to perform parallel mesh adaption, the initial grid must first be partitioned among the available processors. A good partitioner should divide the grid into equal pieces for optimal load balancing, while minimizing the number of edges along partition boundaries for low interprocessor communication. It is also important that within our framework, the partitioning phase be performed rapidly. There are several excellent heuristic algorithms for solving the NP-hard graph partitioning problem [6]. Since mesh partitioning is beyond the scope of this paper, we will assume that a reasonable partition for our test meshes is 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 [2].

3.1 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 number to each element. 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 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 list of shared processors (SPL) is also generated for each shared edge. Finally, the element sublist 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 number on each partition. Next the local edge sublist for each vertex is created from the appropriate subset of the global `edges` array. Like shared edges, each shared vertex must be assigned its SPL. A naive approach would be to thread through the data structures to the elements and their partitions to determine shared vertices. A faster approach is based on the following two properties of a shared vertex: it must be an endpoint 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. An example is shown in Fig. 3 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. For each vertex containing a shared edge in its `edges` sublist, 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.

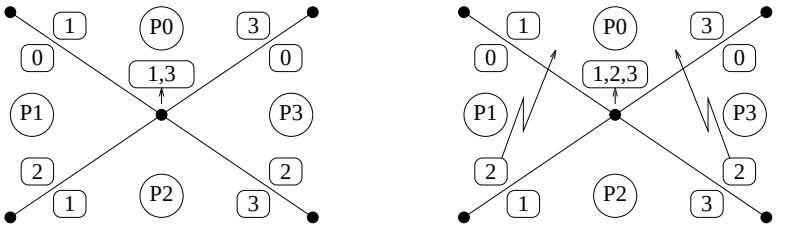


Fig. 3. Example showing the communication need to form the SPL for a shared vertex

The final step in the initialization phase is the local renumbering of the external boundary faces. 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, an edge on the external boundary is shared by exactly two boundary faces. However, when the mesh is partitioned, this is no longer true. An affected edge creates an empty ghost boundary face in each of the two processors for the execution phase which is later eliminated during the finalization stage.

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, 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.

3.2 Execution

The first step in the actual mesh adaption 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 flow solution. This strategy results in a symmetrical marking of all shared edges across partitions since 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 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 4 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. 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.

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 on partition boundaries during refinement are assigned shared processor information. If a shared edge is bisected,

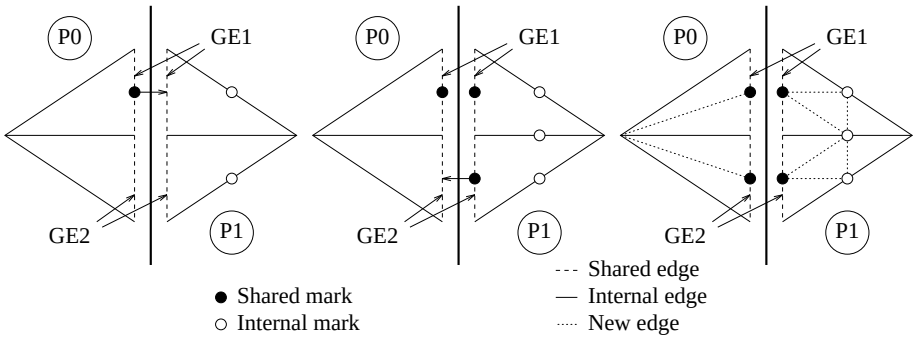


Fig. 4. Two-dimensional example showing communication during propagation of edge

its two children and the center vertex inherit its SPL. However, if a new edge is created that lies across an element face, communication is sometimes required to determine whether it is shared or internal. If it is shared, the SPL must be formed. 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 5 depicts the top view of a tetrahedron in processor P0 that shares two faces with P1. In P0, the intersection of the shared processor lists for the two end-points of each of 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.

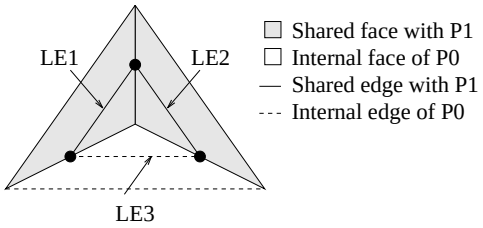


Fig. 5. Example showing how a new edge across a face is classified as shared or internal

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.

3.3 Finalization

Under certain conditions, it is 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. Our finalization phase accomplishes this goal by connecting the subgrids into one global data structure. Individual processors are responsible for correctly arranging the data so that a host processor only collects and concatenates without further processing.

Each local object is first assigned a unique global number. Because elements are not shared, each processor can assign the final global element number by performing a scan-reduce add on the total number of elements. 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 (or vertex) is assigned an owner from its processor list 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 (or vertices), a communication phase propagates the global values from owners to other processors that have local copies.

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 sublists in `elems` cannot be discarded for the unowned edges. Some communication is required to adjust the pointers in the local sublists so that global sublists 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 sublists in `edges` are handled. Since shared vertices may contain local copies of the same global edge in their sublists on different processors, the unowned edge copies are first deleted. Pointers are next adjusted as in the `elems` case with some communication among processors. A final gather operation by the host processor generates the global mesh.

4 Results

The parallel 3D_TAG procedure has been implemented on the SP2 distributed-memory multiprocessor located at NASA Ames Research Center. The code is written in C and C++, with the parallel activities in MPI for portability.

The computational mesh is the one used to simulate the acoustics experiment of Purcell [3] 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 [5]. This paper reports only on the performance of the parallel version of the mesh adaption code.

Timings for the parallel code are presented for one refinement and one coarsening step using various marking and load-balancing strategies. Two marking strategies are used for the refinement step. The first set of experiments consists of randomly marking 5% and 10% of the edges, while the second set consists of marking the same numbers of edges in a single compact region of the mesh. In general, we expect real marking patterns to lie somewhere in between these two significantly different scenarios. Since the coarsening procedure and performance are similar to the refinement method, only one case is presented where 35% of the edges of the largest mesh obtained after refinement are randomly coarsened.

Table 1 presents the progression of grid sizes through the two adaption steps for each marking strategy. Notice that the meshes obtained after refinement for the randomly-marked cases are much larger than those for the locally-marked cases even though exactly the same number of edges are marked. This is due to the difference in the propagation of edge markings. The random cases cause significantly more propagation since refinement is scattered throughout the mesh. The local cases, on the other hand, cause propagation only at the boundary between the refined and the unrefined regions since all edges internal to the refined region are already marked.

Table 1. Progression of grid sizes through refinement and coarsening

		Vertices	Elements	Edges	Bdy Faces
Refinement	Initial mesh	13,967	60,968	74,343	16,818
	5% random marking	24,293	114,415	143,011	8,550
	5% local marking	17,920	82,259	104,178	7,999
	10% random marking	54,389	284,086	345,425	13,606
	10% local marking	21,851	103,582	129,976	8,962
Coarsening	Initial mesh	54,389	284,086	345,425	13,606
	35% random marking	25,689	122,850	152,853	8,630

4.1 Refinement Phase

Table 2 presents the timings and parallel speedup for the refinement step with the random marking of edges. The performance is excellent with efficiencies of almost 90% on 32 processors and 60% to 73% on 64 processors. Notice that the communication time is less than 10% of the total time for up to 16 processors. On 32 and 64 processors, the communication time although still quite small, becomes comparable to the computation time and begins to adversely affect the parallel speedup. This indicates that the saturation point has been reached for this example in terms of the number of processors that should be used. For

example, on 64 processors, each partition contains less than 1,000 elements with 31% 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. Parallel mesh refinement when 10% of the edges are marked shows better performance than the 5%-marked case due to a bigger computation-to-communication ratio. In general, performance will improve as the problem size increases. This is because the computational time will increase while the percentage of elements along processor boundaries will decrease.

Table 2. Performance for the refinement step when edges are marked randomly

		5% Marked			10% Marked		
# Procs	% Shared Edges	Comp Time	Comm Time	Total Speedup	Comp Time	Comm Time	Total Speedup
1	0.0	12.941	0.000	1.00	39.237	0.000	1.00
2	3.2	6.652	0.090	1.92	19.698	0.045	1.99
4	12.1	3.659	0.094	3.45	10.091	0.105	3.85
8	23.2	1.927	0.107	6.36	5.245	0.281	7.10
16	23.9	0.952	0.100	12.30	2.638	0.233	13.67
32	29.2	0.323	0.129	28.63	1.098	0.287	28.33
64	31.0	0.246	0.091	38.40	0.646	0.189	46.99

Table 3 shows the timings and speedup when edges are marked in a single compact region of the global mesh. The performance is extremely poor, with speedups of only 5.1X and 7.7X on 64 processors. This is because we are simulating an almost worst case load balance behavior. This strategy primarily targets elements on one processor only. Most of the other processors remain idle, since none of their elements need to be refined. Noticeable speedup is achieved only when using at least 16 processors. This is because the refinement region remains confined to only one partition until enough processors are used. Once the refinement region straddles multiple partitions, parallelization becomes effective. However, the computation time does decrease somewhat for up to 8 processors, even though all the work is performed by a single processor. This is due to the reduction in the local mesh size for each individual partition. As a result, even though one partition is performing all the work, it has a smaller number of elements to process.

Due to the poor parallel performance when edges are marked in a single compact region of the global mesh, it is worthwhile to load balance the mesh adaption code based on the distribution of targeted edges before these edges are actually refined. The mesh is repartitioned if the markings are skewed beyond a specified tolerance. This significantly improves the performance of the mesh refinement phase. As a bonus, a more balanced mesh is generated after the refinement since the final grid is generally determined by the marking patterns.

Using this methodology, the localized-marking experiment was run again after performing a repartitioning step based on edge markings. A simple heuris-

Table 3. Performance for the refinement step when edges are marked in a single compact region of the global mesh

		5% Marked			10% Marked		
# Procs	% Shared Edges	Comp Time	Comm Time	Total Speedup	Comp Time	Comm Time	Total Speedup
1	0.0	5.581	0.000	1.00	8.806	0.000	1.00
2	3.2	4.351	0.000	1.28	7.517	0.000	1.17
4	12.1	3.828	0.006	1.46	7.036	0.008	1.25
8	23.2	3.362	0.008	1.66	6.462	0.008	1.36
16	23.9	3.230	0.012	1.72	4.232	0.012	2.07
32	29.2	0.982	0.710	3.30	1.188	0.955	4.11
64	31.0	1.083	0.021	5.06	1.104	0.044	7.67

tic of assigning an additional weight to elements containing edges that have been marked for refinement was given to the partitioner. Table 4 presents the performance results of this repartitioned local refinement phase. The communication times are not reported but are considered when calculating the total speedup. Note that the parallel speedups are now comparable to those for the random-marking case. This demonstrates that mesh adaption can deliver excellent speedups if the marked edges are equidistributed among the processors.

Table 4. Performance for the repartitioned refinement step when edges are marked in a single compact region of the global mesh

	5% Marked				10% Marked			
# Procs	# Elements in Min Set	Max Set	Comp Time	Total Speedup	# Elements in Min Set	Max Set	Comp Time	Total Speedup
1	60,968	60,968	5.581	1.00	60,968	60,968	8.806	1.00
2	9,069	51,899	2.486	1.72	6,867	54,101	3.977	1.80
4	5,575	28,983	1.446	3.44	3,074	42,701	2.376	3.47
8	2,120	14,498	0.824	6.62	1,272	21,358	1.244	6.89
16	389	7,249	0.287	12.19	595	10,670	0.622	12.91
32	190	3,629	0.251	21.22	281	5,340	0.352	24.26
64	95	1,812	0.132	36.24	141	2,670	0.147	43.59

4.2 Coarsening Phase

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

Timings and parallel speedup when 35% of the edges of the largest mesh obtained by refinement are randomly coarsened are presented in Table 5. Note

that the computation time does not include the follow-up mesh refinement time. It is, instead, only the time required to mark edges to coarsen. This was done so as to demonstrate the parallel performance of the modules that are only required during the coarsening phase. Notice that the communication time is negligible while the cleanup time is dominant. Since the cleanup time depends on the fraction of shared objects, performance deteriorates as the problem size is over-saturated by processors.

Table 5. Performance for the coarsening step when edges are marked randomly

# Procs	Comp Time	Cleanup Time	Comm Time	Total Speedup
1	3.184	6.949	0.000	1.00
2	1.648	3.564	0.005	1.94
4	0.850	1.822	0.006	3.78
8	0.439	0.962	0.011	7.18
16	0.270	0.499	0.024	12.78
32	0.144	0.271	0.020	23.29
64	0.085	0.132	0.038	39.74

4.3 Initialization and Finalization Phases

Recall from Fig. 1 that unlike the execution phase where the actual adaption is performed, it is not critical for the initialization and finalization procedures to be very efficient since they are used rarely (or only once) during a flow computation. Table 6 presents the results for these two phases. The initialization step is performed on the starting mesh consisting of 60,968 elements, while the finalization phase is for the adapted mesh consisting of 114,415 elements. It is apparent from the timings that the performance bottleneck for the two steps are the global broadcast (one-to-all) and gather (all-to-one) communication patterns, respectively. These times generally increase with the number of processors

Table 6. Performance for the initialization and finalization steps when 5% of edges are marked randomly

	Initialization			Finalization		
# Procs	Comp Time	Bcast Time	Total Speedup	Comp Time	Gather Time	Total Speedup
1	4.500	0.328	1.00	4.035	0.682	1.00
2	2.479	0.645	1.55	2.312	0.665	1.58
4	1.523	1.175	1.79	1.494	0.676	2.17
8	0.962	0.918	2.57	1.019	0.714	2.72
16	0.568	1.008	3.06	0.647	0.785	3.29
32	0.409	1.214	2.97	0.393	0.890	3.68
64	0.242	1.503	2.77	0.286	0.977	3.73

so a speedup cannot be expected. However, the computational sections of these procedures do show favorable speedups of 18.6X and 14.1X on 64 processors. In any case, the overall run times of these routines are acceptable for our purposes. Note that the broadcast and gather times are non-zero even for a single processor because the current implementation uses a host to perform the data I/O. The number of processors shown in Table 6 indicates those that are actually performing the mesh adaption.

5 Summary

Fast and efficient dynamic mesh adaption is an important feature of unstructured grids that make them especially attractive for unsteady flows. For such flows, the coarsening/refinement step must be performed frequently, so its efficiency must be comparable to that of the flow solver. For this work, the adaption scheme of Biswas and Strawn [1] is parallelized for distributed-memory architectures.

The code consists of approximately 3,000 lines of C++ with MPI which wrap around the original version written in C. The serial code was left almost completely unchanged except for the addition of 10 lines which interface to the parallel wrapper. This allowed us to design the parallel version using the serial code as a building block. 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 for objects on partition boundaries.

Parallel performance is extremely promising showing a 47-fold speedup on 64 processors compared to sequential execution. In the worst case when a single compact region of the mesh is refined, speedup increased from 8- to 44-fold by repartitioning the mesh using the edge-marking information. We are currently in the process of combining this parallel mesh adaption code with a dynamic partitioner and load balancer [4].

References

1. Biswas, R., Strawn, R.: A new procedure for dynamic adaption of three-dimensional unstructured grids. Appl. Numer. Math. **13** (1994) 437–452
2. Hendrickson, B., Leland, R.: The Chaco user's guide — Version 2.0. Sandia National Laboratories Technical Report SAND94-2692 (1994)
3. Purcell, T.: CFD and transonic helicopter sound. 14th European Rotorcraft Forum (1988) Paper 2
4. Sohn, A., Biswas, R., Simon, H.: A dynamic load balancing framework for unstructured adaptive computations on distributed-memory multiprocessors. 8th ACM Symposium on Parallel Algorithms and Architectures (1996) to appear
5. Strawn, R., Biswas, R., Garceau, M.: Unstructured adaptive mesh computations of rotorcraft high-speed impulsive noise. J. Aircraft **32** (1995) 754–760
6. Van Driessche, R., Roose, D.: Load Balancing Computational Fluid Dynamics Calculations on Unstructured Grids. AGARD Report R-807 (1995)

This article was processed using the LaTeX macro package with LLNCS style