

Parallel Mesh Adaption with Global Load Balancing on the SP2

Rupak Biswas
MRJ, Inc., NASA ARC
Moffett Field, CA 94035
(415) 604-4411
rbiswas@nas.nasa.gov

Leonid Oliker
RIACS, NASA ARC
Moffett Field, CA 94035
(415) 604-4316
oliker@riacs.edu

Andrew Sohn
CIS Dept., NJIT
Newark, NJ 07102
(201) 596-2315
sohn@cis.njit.edu

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. Unfortunately, the adaptive solution of unsteady problems causes load imbalance among processors on a parallel machine because the computational intensity is both space and time dependent. This requires significant communication at runtime leading to idle processors and adversely affecting the total execution time. Various methods on dynamic load balancing have been reported to date; however, most of them lack a global view of loads across processors.

Figure 1 depicts our framework for parallel adaptive flow computation. The mesh is first partitioned and mapped among the available processors. A flow solver then runs for several iterations, updating solution variables. Once an acceptable solution is obtained, a mesh adaption procedure is invoked to generate a new computational mesh based on an error indicator. A quick evaluation step determines if the new mesh is sufficiently unbalanced to warrant a repartitioning. If the current partitions are adequately load balanced, control is passed back to the flow solver. Otherwise, a repartitioning procedure divides 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 remapping cost 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. 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 flow solvers are usually several times more expensive.

This paper briefly describes an efficient parallel implementation of the 3D_TAG tetrahedral mesh adaption scheme that has shown good sequential performance on the C90 when coupled with a variety of unstructured flow solvers to solve realistic problems in rotary- and fixed-wing aerodynamics [1,5]. The parallel version [3] consists of an additional 3000 lines of C++ with MPI, allowing portability to any platform supporting these languages. This code is a wrapper around the original 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.

This paper also describes a new method that has been developed to dynamically balance the processor workloads with a global view [4]. The load-balancing procedure uses a dual graph representation of the initial mesh 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. Although mesh repartitioning is an inherent component of our global load balancing scheme, it is not addressed here. Partitioning results for small model problems in the context of dynamic load balancing are reported in [4].

Parallel Mesh Adaption

Complete details of the 3D_TAG mesh adaption scheme are given in [1]. At each 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 an 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. The 1-to-8 isotropic subdivision is implemented by adding a new vertex at the mid-point of each of the six edges. The 1-to-4 and 1-to-2 subdivisions result either because a tetrahedron is targeted anisotropically or because they are required to form a valid connectivity for the new 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.

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 binary pattern. Elements are continuously upgraded to valid patterns corresponding to the three allowed subdivision types until none of the patterns show any change. 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.

The parallel implementation of the 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↔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 partition 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 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 adapts 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 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. This causes some propagation of edges 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. 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, 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.

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 information is generated since no mesh objects are created during this step. However, objects are renumbered due to 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 process 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. 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.

Global Load Balancing

The dual graph representation of the initial computational mesh is one of the key features of this work. Each dual graph vertex has two weights associated with it. The computational weight, w_{comp} , indicates the workload for the corresponding element. The remapping weight, w_{remap} , indicates the cost of moving the element from one processor to another. The weight w_{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_{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 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_{comp} and w_{remap} , for every element in the initial mesh.

The preliminary evaluation step rapidly determines if the dual graph with a new set of w_{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. If, on the other hand, the loads are unbalanced, the mesh is repartitioned. 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. 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 where 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 . To minimize the total data movement for all processors, 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 $j_i \neq j_m, \forall i \neq m$. Both an optimal and a heuristic greedy algorithm have been implemented for solving this problem. A similarity matrix with the new processor assignment for 8 partitions is shown in Fig. 2. Only the non-zero entries are shown.

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. The redistribution cost is calculated from the similarity matrix S using two machine-dependent parameters: the remote-memory latency time T_{lat} and the message setup time T_{setup} . The new partitioning and mapping are accepted if the computational gain is larger than the redistribution cost.

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.

Results

The parallel 3D_TAG and global load balancing procedures have been implemented on the SP2 located at NASA Ames Research Center. Both codes are written in C and C++, with the parallel activities in MPI for portability. The computational mesh, containing 60,968 tetrahedral elements and 78,343 edges, is one used to simulate an acoustics experiment 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]. Results are presented for one adaption step using two different edge-marking strategies. The first strategy, called `Random`, consisted of randomly targeting 35% of the edges for refinement. The second strategy, called `Local`, refined the same number

of edges in a single compact region of the mesh. Both adaption strategies resulted in a final mesh consisting of more than 201,000 elements and 246,000 edges.

Figure 3 shows the parallel speedup of the refinement phase. As expected, **Random** gives the best performance of 47.0X on 64 processors as the workloads are inherently balanced. Performance deteriorates to 21.8X for the **Local** strategy. This is because almost all the mesh adaption is confined to a small subset of the processors. 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.

Figure 4 shows how the execution time is spent during the refinement and the subsequent load-balancing phases for the **Local** strategy. The repartitioning times are not shown as it is not the focus of this paper. Note that the remapping time initially increases with the number of processors, but then gradually decreases. This is because 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 the remapping phase will not become a bottleneck. Processors were reassigned using our heuristic algorithm. 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.

Figure 5 compares the execution times and the amount of data movement for the **Local** strategy when using the optimal and heuristic processor assignment algorithms. The optimal method requires almost two orders of magnitude more time than our heuristic method. The relative reduction in data movement, however, is not very significant for our test cases.

Figure 6 illustrates the impact of load balancing on the execution time of the flow solver. Note that the maximum possible improvement is not linear. For the 3D-TAG code, load balancing for P processors could give a maximum improvement of $\frac{8P}{P+7}$ over a non-balanced load [2]. The **Random** case gives only a marginal improvement when the processor loads are balanced because the computational work is already distributed uniformly among the processors after mesh adaption. The **Local** strategy shows a much bigger 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 note that the results in Fig. 6 are for a single refinement step. With repeated adaption, the gains realized with load balancing may be even more significant.

References

- [1] R. Biswas and R.C. Strawn, "A New Procedure for Dynamic Adaption of Three-Dimensional Unstructured Grids," *Appl. Numer. Math.*, 13 (1994) 437-452.
- [2] R. Biswas, L. Oliker, and A. Sohn, "Global Load Balancing with Parallel Mesh Adaption on Distributed-Memory Systems," *Supercomputing'96*, to appear.
- [3] L. Oliker, R. Biswas, and R.C. Strawn, "Parallel Implementation of an Adaptive Scheme for 3D Unstructured Grids on the SP2," *Irregular'96*, to appear.
- [4] A. Sohn, R. Biswas, and H. Simon, "Impact of Load Balancing on Unstructured Adaptive Grid Computations for Distributed-Memory Multiprocessors," *SPDP'96*, to appear.
- [5] R.C. Strawn, R. Biswas, and M. Garceau, "Unstructured Adaptive Mesh Computations of Rotorcraft High-Speed Impulsive Noise," *J. of Aircraft*, 32 (1995) 754-760.

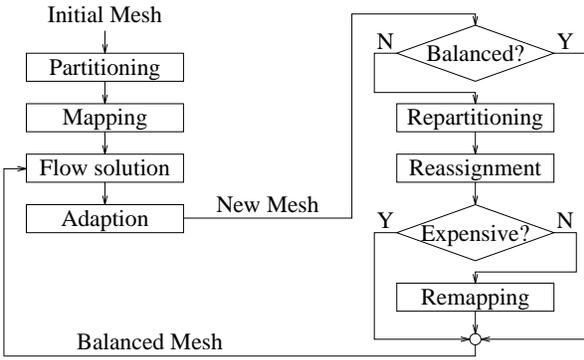


Figure 1: Overview of our framework for parallel adaptive flow computations.

		New Partitions							
		0	1	2	3	4	5	6	7
Old Processors	0	389	510		120				
	1	11		563	444				
	2	236	401			333		47	
	3	129	130		129	146	43	446	
	4					13		490	502
	5								1024
	6						1020		
	7					467	471	92	
		7	0	1	4	2	6	3	5

Figure 2: A similarity matrix after processor reassignment.

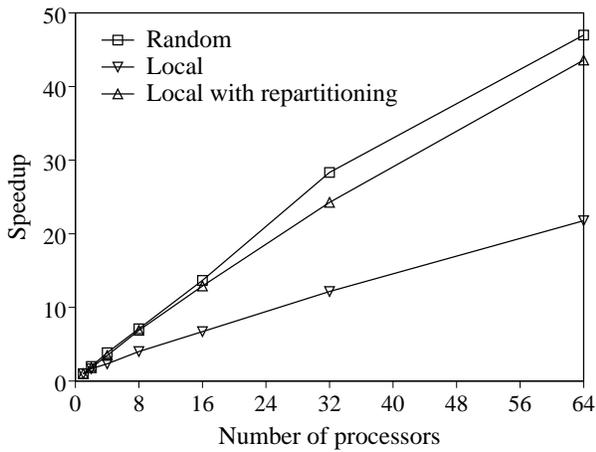


Figure 3: Speedup of the parallel mesh adaptation code during the refinement stage.

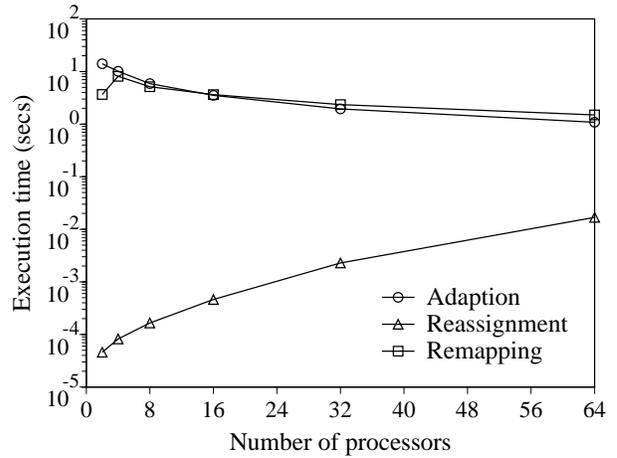


Figure 4: Anatomy of total execution time for the Local refinement strategy.

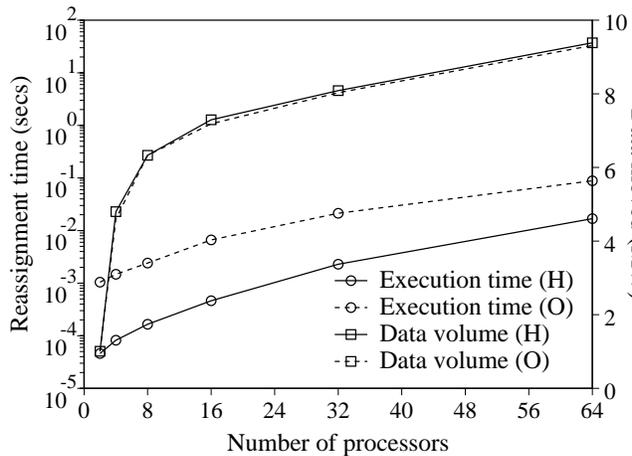


Figure 5: Comparison of the heuristic (H) and optimal (O) mappers for the Local strategy.

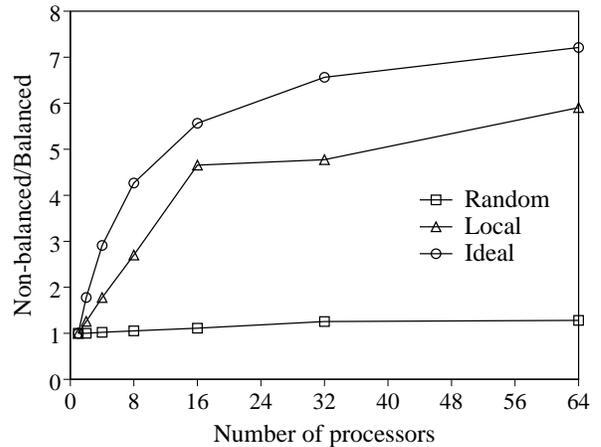


Figure 6: Comparison of flow solver execution times with and without load balancing.