Load Balancing Sequences of Unstructured Adaptive Grids

Rupak Biswas

MRJ Technology Solutions
NASA Ames Research Center
Moffett Field, CA 94035
rbiswas@nas.nasa.gov

Leonid Oliker

RIACS
NASA Ames Research Center
Moffett Field, CA 94035
oliker@riacs.edu

Abstract

Mesh adaption is a powerful tool for efficient unstructured grid computations but causes load imbalance on multiprocessor systems. To address this problem, we have developed PLUM*, an automatic portable framework for performing adaptive large-scale numerical computations in a message-passing environment. This paper makes several important additions to our previous work. First, a new remapping cost model is presented and empirically validated on an SP2. Next, our load balancing strategy is applied to sequences of dynamically adapted unstructured grids. Results indicate that our framework is effective on many processors for both steady and unsteady problems with several levels of adaption. Additionally, we demonstrate that a coarse starting mesh produces high quality load balancing, at a fraction of the cost required for a fine initial mesh. Finally, we show that the data remapping overhead can be significantly reduced by applying our heuristic processor reassignment algorithm.*

1 Introduction

Dynamic mesh refinement and coarsening on unstructured grids is a powerful tool for computing large-scale problems that require grid modifications to efficiently resolve solution features. 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. Various methods on dynamic load balancing have been reported to date; however, most of them lack a global view of loads across processors.

Our goal is to build a portable system for efficiently performing adaptive large-scale numerical calculations in a parallel message-passing environment. Figure 1 depicts our framework, called PLUM, for such an automatic system. The mesh is first partitioned and mapped among the available processors. A solver then runs for several iterations, updating solution variables. Once an acceptable solution is obtained, a mesh adaption procedure is invoked. It first targets edges for coarsening and refinement based on an error indicator computed from the numerical solution. The old mesh is then coarsened, resulting in a smaller grid. Since edges have already been marked for refinement, it is possible to exactly predict the new mesh before actually performing the refinement step. Program control is thus passed to the load balancer at this time. A quick evaluation step determines if the new mesh will be so unbalanced as to warrant a repartitioning. If the current partitions will remain adequately load balanced, control is passed back to the subdivision phase of the mesh adaptor. Otherwise, a repartitioning procedure is used 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 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. The computational mesh is then actually refined and the numerical calculation is restarted.

Extensive details of the parallel mesh adaption scheme, called 3D_TAG, that is used in this work is given in [6]. The parallel version consists of C++ and MPI code wrapped around the original serial mesh adaption program [3]. An object-oriented approach allowed the distributed-memory implementation to be performed in a clean and efficient manner. Notice from the framework in Fig. 1 that splitting the mesh refinement step into two distinct phases of edge marking and mesh subdivision allows the subdivision phase to operate in a more load balanced fashion. In addition, since data remapping is performed before the mesh grows in size due to refinement, a smaller volume of data is moved. This, in turn, leads to significant savings in the redistribution cost.

2 Dynamic Load Balancing

PLUM is a novel method to dynamically balance

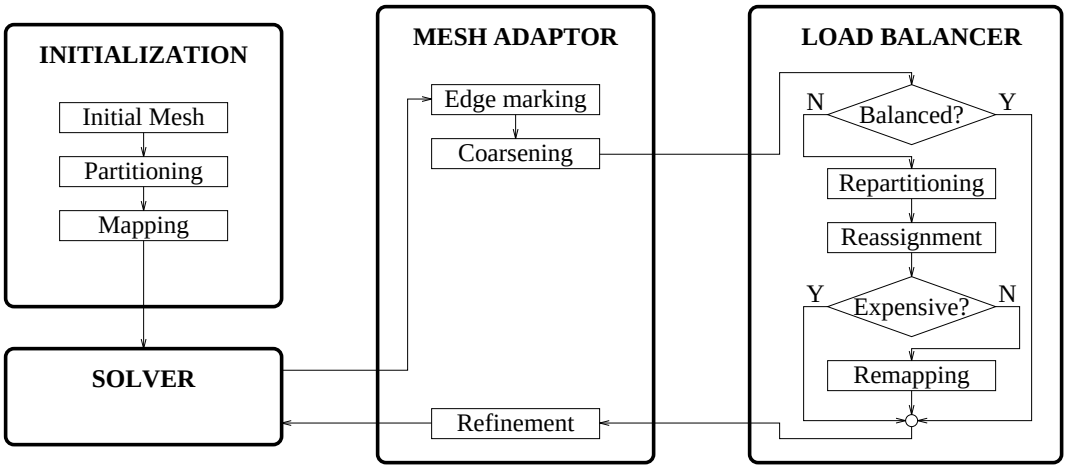

Figure 1: Overview of PLUM, our framework for parallel adaptive numerical computation.

the processor workloads with a global view. Results reported earlier either focused on fundamental load balancing issues [7] or various refinement strategies [2,5] to demonstrate the viability and effectiveness of our framework. This paper presents, for the first time, the application of PLUM to sequences of dynamically adapted unstructured grids. We also present a data remapping cost model that can accurately predict the total cost of data redistribution on an SP2 given the number of tetrahedral elements that have to be moved among the processors.

Our load balancing procedure has five novel features: (i) a dual graph representation of the initial computational mesh keeps the complexity and connectivity constant during the course of an adaptive computation; (ii) a parallel mesh repartitioning algorithm avoids a potential serial bottleneck; (iii) a heuristic remapping algorithm quickly assigns partitions to processors so that the redistribution cost is minimized; (iv) an efficient data movement scheme allows remapping and mesh subdivision at a significantly lower cost than previously reported; and (v) accurate metrics estimate and compare the computational gain and the redistribution cost of having a balanced workload after each mesh adaption step.

Using the dual of the initial computational mesh for the purpose of dynamic load balancing is one of the key features of this work. Each dual graph vertex has two weights associated with it. The computational weight, w_{comp}, models the workload for the corresponding element. The remapping weight, w_{remap}, models 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 numerical 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. Every edge of the dual graph also has a weight w_{comm} that models the runtime interprocessor communication. The value of w_{comm} is set to the number of faces in the computational mesh that corresponds to the dual graph edge. The mesh connectivity, w_{comp}, and w_{comm} together determine how dual graph vertices should be grouped to form partitions that minimize both the disparity in the partition weights and the runtime communication. The w_{remap} determines how partitions should be assigned to processors such that the cost of data redistribution is minimized. New computational grids obtained by adaption are translated to w_{comp} and w_{remap} for every vertex and to w_{comm} for every edge in the dual mesh.

If a preliminary evaluation step determines that the dual graph with a new set of w_{comp} is unbalanced, the mesh needs to be repartitioned. A good partitioner should minimize the total execution time by balancing the computational loads and reducing the interprocessor communication time. In addition, the repartitioning phase must be performed very rapidly for our load balancing framework to be viable. For the test cases in this paper, an alpha version of ParMeTiS [4], a parallel multilevel algorithm, was used as the repartitioner. Results indicate that this partitioner can be effectively used inside PLUM; however, any other algorithm can also be used as long as it quickly delivers partitions that are reasonably balanced and require minimal communication.

Once new partitions are obtained, they must be mapped to processors such that the redistribution cost is minimized. In general, the number of new partitions is an integer multiple F of the number of processors.

Each processor is then assigned F unique partitions. 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 $S_{i,j}$ is the sum of the w_{remap} of all the dual graph vertices in new partition j that already reside on processor i. A similarity matrix for $P = 4$ and $F = 2$ is shown in Fig. 2. Only the non-zero entries are shown.

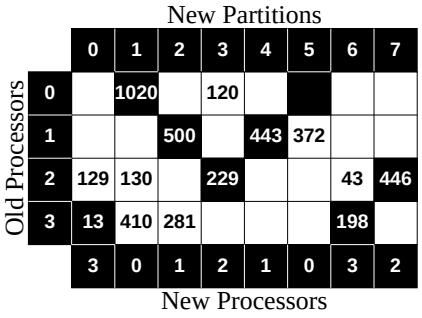

Figure 2: A similarity matrix after processor reassignment using the heuristic algorithm and the `TotalV` metric.

The goal of the processor reassignment phase is to find a mapping between partitions and processors such that the data redistribution cost is minimized. Various cost functions are usually needed to solve this problem for different architectures. In [5], we investigated two general metrics: `TotalV`, that minimizes the total volume of data moved among all processors, and `MaxV`, that minimizes the maximum flow of data to or from any single processor. `TotalV` assumes that by reducing network contention and the total number of elements moved, the remapping time will be reduced. Both an optimal and a heuristic greedy algorithm have been implemented for solving the processor reassignment problem using `TotalV` [5]. Applying the heuristic procedure to the similarity matrix in Fig. 2 generates the processor assignment shown in the bottom row. It was proved in [5] that a processor assignment obtained using the heuristic algorithm can never result in a data movement cost that is twice that of the optimal assignment. `MaxV`, on the other hand, considers data redistribution in terms of solving a load imbalance problem, where it is more important to minimize the workload of the most heavily-weighted processor than to minimize the sum of all the loads. An optimal algorithm for solving the assignment problem using `MaxV` has also been implemented [5].

3 Remapping Cost Model

Once the reassignment problem is solved, a model is needed to quickly predict the expected redistribution cost for a given architecture. Accurately estimating this time is very difficult due to the large number and complexity of the costs involved in the remapping procedure. The computational overhead includes rebuilding internal data structures and updating shared boundary information. Predicting the latter cost is particularly challenging since it is a function of the old and new partition boundaries. The communication overhead is architecture-dependent and can be difficult to predict especially for the many-to-many collective communication pattern used by the remapper.

Our redistribution algorithm consists of three major steps: first, the data objects moving out of a partition are stripped out and placed in a buffer; next, a collective communication appropriately distributes the data to its destination; and finally, the received data is integrated into each partition and the boundary information is consistently updated. Performing the remapping in this bulk fashion, as opposed to sending individual small messages, has several advantages including the amortization of message start up costs and good cache performance. Additionally, the total time can be modeled by examining each of the three steps individually since the two computational phases are separated by the implicit barrier synchronization of the collective communication. The computation time can therefore be approximated as:

$$\alpha \times \max(\texttt{ElemsSent}) + \beta \times \max(\texttt{ElemsRecd}) + \delta,$$

where α and β represent the time necessary to strip out and insert an element respectively, and δ is the additional cost of processing boundary information. The maximum values of `ElemsSent` and `ElemsRecd` can be quickly derived from the solved similarity matrix. Since the value of δ is difficult to predict exactly and constitutes a relatively small part of the computation, we assume that it is a small constant. To simplify our model even further, we assume that $\alpha = \beta$.

Much work has been done to model communication overhead including LogGP [1] and BSP [9]. Both models make the following assumptions which hold true for most architectures including the SP2: a receiving processor may access a message or parts of it only after the entire message has arrived; and, at any given time a processor can either be sending or receiving a single message. Our redistribution procedure closely follows the superstep model of BSP. An advantage of the SP2 interconnection mechanism is that all nodes can be considered equidistant from one another. This allows us to predict communication overhead without the need to model multiple hops for individual messages. We approximate our communication cost as:

$$g \times \max(\texttt{ElemsSent}) + g \times \max(\texttt{ElemsRecd}) + l,$$

where g is a machine-specific cost of moving a single element and l is the time for barrier synchronization.

The total expected time for the redistribution procedure can therefore be expressed as:

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

where $\texttt{MaxSR} = \max(\texttt{ElemsSent}) + \max(\texttt{ElemsRecd})$, $\gamma = \alpha + g$, and $O = \delta + l$. In order to compute the slope and intercept of this linear function, several data points need to be generated for various redistribution patterns and their corresponding run times. A simple least squares fit can then be used to approximate γ and O. This procedure needs to be performed only once for each architecture, and the values of γ and O can then be used in actual computations to estimate the redistribution cost. Note that there is a close relationship between `MaxSR` of the remapping cost model and the theoretical metric `MaxV`. The optimal similarity matrix solution for `MaxSR` is provably no more than twice that of `MaxV`.

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. It can be expressed as $T_{\text{iter}} N_{\text{adapt}} (W_{\max}^{\text{old}} - W_{\max}^{\text{new}})$, where T_{iter} is the time required to run one solver iteration on one element of the original mesh, N_{adapt} is the number of solver iterations between mesh adaptions, and $W_{\max}^{\text{old}}$ 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. The new partitioning and processor reassignment are accepted if the computational gain is larger than the redistribution cost. The numerical simulation is then interrupted to redistribute the data.

4 Results

The 3D_TAG parallel mesh adaption procedure and the PLUM global load balancing strategy have been implemented in C, C++, and MPI on an SP2. The computational mesh for the test cases in this paper 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. Detailed numerical results of the simulation are given in [8]. A cut-out view of the initial tetrahedral mesh is shown in Fig. 3.

In the **first set of experiments**, a total of three adaptions are performed in sequence on this initial mesh. Table I lists the size of the computational mesh after each level of adaption. Notice that the final mesh is more than an order of magnitude larger than the initial mesh. This is a steady-state calculation where mesh adaption is used to resolve the leading edge compression and capture both the surface shock and the acoustic wave that propagates to the far field.

Figure 4 shows how the execution time is spent during the adaption and the subsequent load balancing

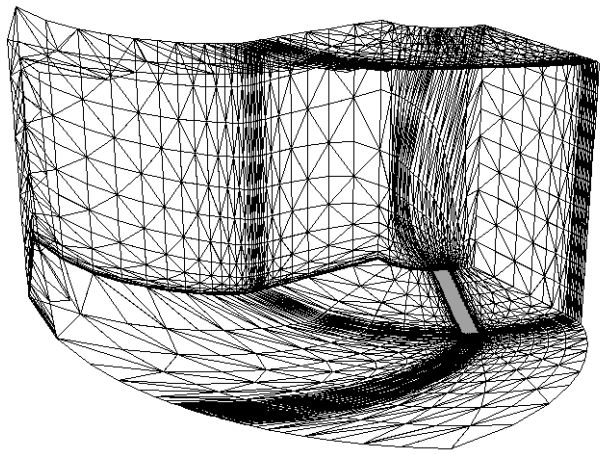

Figure 3: Cut-out view of the initial tetrahedral mesh.

	Vertices	Elements	Edges
Initial	13,967	60,968	78,343
Level 1	35,219	179,355	220,077
Level 2	72,123	389,947	469,607
Level 3	137,474	765,855	913,412

Table I: Progression of grid size through a sequence of three levels of adaption for a steady-state computation.

phases for the first and the third levels. Our heuristic greedy algorithm is used to perform the processor reassignment. The reassignment times are not shown since they are several orders of magnitude smaller than the other times. The repartitioning curves, using ParMeTiS [4], are almost identical for the three levels because the time to repartition mostly depends on the initial problem size. The repartitioning times are also almost independent of the number of processors. The mesh adaption times increase with the size of the mesh; however, they consistently show an efficiency of about 85% on 64 processors for all three levels. In fact, the efficiency increases with the mesh size because of a larger computation-to-communication ratio. The remapping times gradually decrease as the number of processors is increased. 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. The remapping time increases from one adaption level to the next because of the growth in the mesh size. However, as shown later in this paper, the remapping times stabilize when the mesh size remains approximately constant. More importantly, the remapping times always dominate and are generally about four times the adaption time on 64 processors. This is not unexpected since remapping is considered *the* bottleneck in dynamic load balancing. It is for this reason that the remapping cost needs to

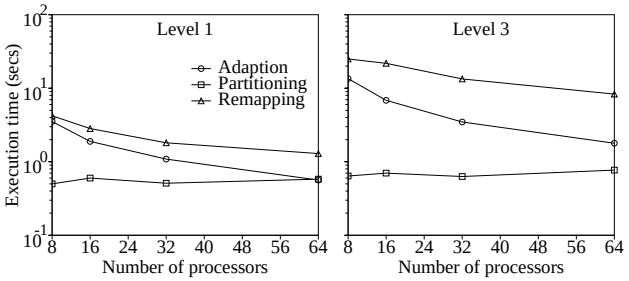

Figure 4: Execution times for the three levels of adaption.

be predicted accurately before a load balancing phase to be certain that the data redistribution cost will be more than compensated by the computational gain.

The **second set of experiments** is performed to compute the slope γ and the intercept O of our redistribution cost model. Experimental data is gathered by running various redistribution patterns. The remapping times are then plotted against two metrics, `TotalV` and `MaxSR`, in Fig. 5. Results demonstrate that on an SP2, there is little obvious correlation between the total number of elements moved (`TotalV`) and the expected run time for the remapping procedure. On the other hand, there is a clear linear correlation between the maximum number of elements moved (`MaxSR`) and the actual redistribution time. There are some perturbations in the plots resulting from factors such as network hotspots and shared data irregularities, but the overall results show that our redistribution model successfully estimates the data remapping time. This important result indicates that reducing the bottleneck, rather than the aggregate, overhead guarantees a reduction in the redistribution time.

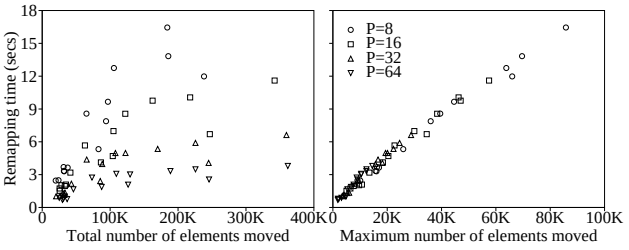

Figure 5: Remapping time as a function of `TotalV` and `MaxSR` metrics.

The **third set of experiments** is performed to evaluate the performance of PLUM in an unsteady environment where the adapted region is strongly time-dependent. To achieve this goal, a simulated shock wave is propagated through the initial mesh shown in Fig. 3. The test case is generated by refining all elements within a cylindrical volume moving left to right across the domain with constant velocity, while coarsening previously-refined elements in its wake. The performance of PLUM is measured at nine successive adaption levels. Note that because these results are derived directly from the dual graph, mesh adaption times are not reported, and remapping overheads are computed using our redistribution cost model.

Figure 6 shows the progression of grid size for the nine levels of adaption in the unsteady simulation. Both coarse and fine meshes are used in the experiment to investigate the relationship between load balancing performance and dual graph size. The initial fine mesh is eight times the size of the coarse mesh shown in Fig. 3. Note that although an axisymmetric cylinder moves through the meshes at constant velocity, the sizes of the meshes change erratically due to the nonuniformity of the initial meshes.

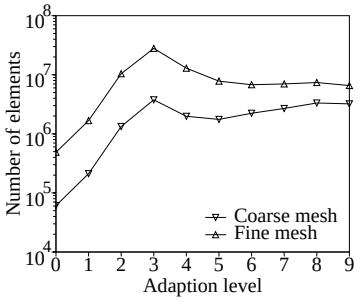

Figure 6: Progression of grid size through nine levels of adaption for an unsteady computation.

Figure 7 presents the partitioning and remapping times for both mesh granularities. Two remapping strategies are used, resulting in different remapping times at each level. One strategy uses the default processor mapping given by ParMeTiS [4], while the other performs processor reassignment based on our heuristic solution of the similarity matrix. Several observations can be made from the resulting graphs. First, our heuristic remapper always outperforms the default strategy, typically resulting in over a two-fold speedup of the data remapping phase. This shows that processor reassignment must be performed using a proper metric to minimize the remapping time. Second, when comparing dual graph granularities, results show that

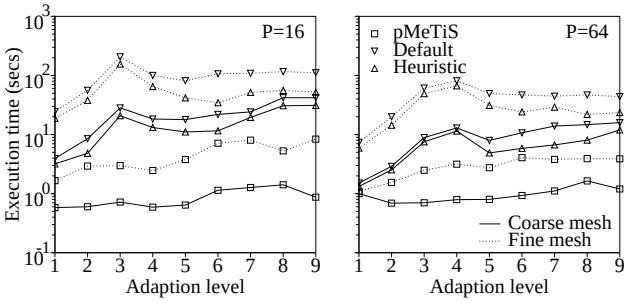

Figure 7: Partitioning and remapping times.

the finer mesh increases both the partitioning and the remapping times by almost an order of magnitude. This is expected since the larger graph is harder to partition and requires more data movement during remapping. Finally, increasing the number of processors does not have a major effect on the partitioning overhead, but causes a noticeable reduction in the remapping times. This indicates that our load balancing strategy will remain viable on a large number of processors.

Figure 8 presents the quality of load balancing for both meshes. Load balancing quality is defined in two ways: the computational load imbalance factor and the percentage of cut edges. The load imbalance factor is the ratio of the sum of the w_{comp} on the most heavily-loaded processor to the average load across all processors. For all the cases, the partitioner does an excellent job of reducing the imbalance factor to unity. Using a finer mesh has a negligible effect on the imbalance factor after load balancing, but requires a substantially longer repartitioning time. The percentage of cut edges always increases with the number of processors. This is expected since the surface-to-volume ratio increases with the number of partitions. Notice that the percentage of cut edges generally grows with each level of adaption. This is because successive adaptions create a complex distribution of computationally-heavy nodes in the dual graph, thereby requiring partitions to have more complicated boundaries to achieve load balance. This increases the surface-to-volume ratio of the partitions, resulting in a higher percentage of cut edges. The finer mesh consistently has a smaller percentage of cut edges because the partitioner has a wider choice of edges to find a better cut. However, we believe that this savings in the number of cut edges does not warrant the significantly higher overhead of the finer mesh.

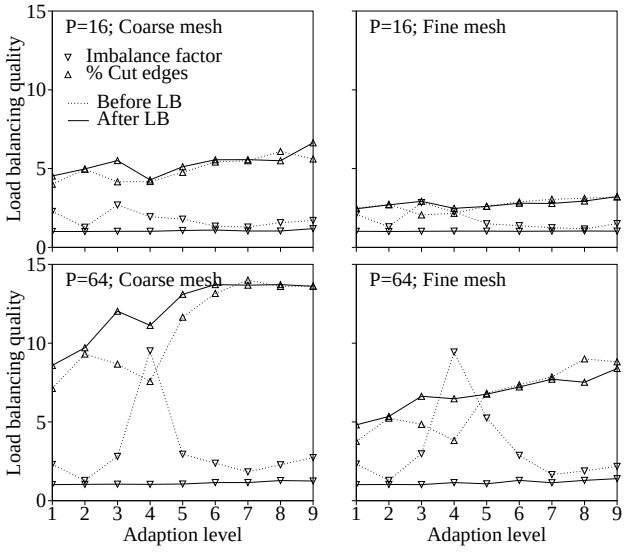

Figure 8: Quality of load balancing.

5 Conclusions

We have shown in this paper that our load balancing scheme, called PLUM, works well for both steady and unsteady adaptive problems with many levels of adaption, even when using a coarse initial mesh. A finer starting mesh may be used to achieve lower edge cuts and marginally better load balance, but is generally not worth the increased partitioning and data remapping times. Results have also demonstrated that our framework scales with the number of processors and that our heuristic processor reassignment algorithm significantly reduces data remapping times. Finally, a new remapping cost model was presented and quantitatively validated. Results indicated that reducing the bottleneck overhead guarantees a reduction in the total redistribution time.

References

[1] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model — One Step Closer towards a Realistic Model for Parallel Computation," *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*, Santa Barbara, CA, 1995, pp. 95–105.

[2] R. Biswas, L. Oliker, and A. Sohn, "Global Load Balancing with Parallel Mesh Adaption on Distributed-Memory Systems," *Proc. Supercomputing'96*, Pittsburgh, PA, 1996.

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

[4] G. Karypis and V. Kumar, "Parallel Multilevel K-way Partitioning Scheme for Irregular Graphs," Report 96-036, University of Minnesota, 1996.

[5] L. Oliker and R. Biswas, "Efficient Load Balancing and Data Remapping for Adaptive Grid Calculations," *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures*, Newport, RI, 1997, pp. 33–42.

[6] L. Oliker, R. Biswas, and R. Strawn, "Parallel Implementation of an Adaptive Scheme for 3D Unstructured Grids on the SP2," *Parallel Algorithms for Irregularly Structured Problems*, LNCS 1117, Springer-Verlag, 1996, pp. 35–47.

[7] A. Sohn, R. Biswas, and H. Simon, "Impact of Load Balancing on Unstructured Adaptive Grid Computations for Distributed-Memory Multiprocessors," *Proc. 8th IEEE Symp. on Parallel and Distributed Processing*, New Orleans, LA, 1996, pp. 26–33.

[8] R. Strawn, R. Biswas, and M. Garceau, "Unstructured Adaptive Mesh Computations of Rotorcraft High-Speed Impulsive Noise," *J. Aircraft* 32 (1995) 754–760.

[9] L. Valiant, "A Bridging Model for Parallel Computation," *Comm. of the ACM*, 33 (1990) 103–111.