PLUM: Parallel Load Balancing for Adaptive Unstructured Meshes[1]

Leonid Oliker
Research Institute for Advanced Computer Science
Mail Stop T27A-1, NASA Ames Research Center
Moffett Field, California 94035-1000
oliker@riacs.edu

Rupak Biswas
MRJ Technology Solutions
Mail Stop T27A-1, NASA Ames Research Center
Moffett Field, California 94035-1000
rbiswas@nas.nasa.gov

ABSTRACT

Mesh adaption is a powerful tool for efficient unstructured-grid computations but causes load imbalance among processors on a parallel machine. We present a novel method called PLUM to dynamically balance the processor workloads with a global view. This paper presents the implementation and integration of all major components within our dynamic load balancing strategy for adaptive grid calculations. Mesh adaption, repartitioning, processor assignment, and remapping are critical components of the framework that must be accomplished rapidly and efficiently so as not to cause a significant overhead to the numerical simulation. A data redistribution model is also presented that predicts the remapping cost on the SP2. This model is required to determine whether the gain from a balanced workload distribution offsets the cost of data movement. Results presented in this paper demonstrate that PLUM is an effective dynamic load balancing strategy which remains viable on a large number of processors.

1. 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 refined meshes are required to accurately capture shock waves, contact discontinuities, vortices, and

[1]This work was supported by NASA under Contract Number NAS 2-96027 with the Universities Space Research Association and under Contract Number NAS 2-14303 with MRJ Technology Solutions.

shear layers. Local mesh adaption provides the opportunity to obtain solutions 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 both space and time dependent. This requires significant communication at runtime leading to idle processors and adversely affecting the total execution time. An efficient parallel implementation of such methods is extremely difficult to achieve, primarily because of the dynamically-changing nonuniform grid. Various methods on dynamic load balancing have been reported to date [7,8,10-12,14-17,19,20,22,33-35]; however, most of them either lack a global view of loads across processors or do not apply their techniques to realistic large-scale applications.

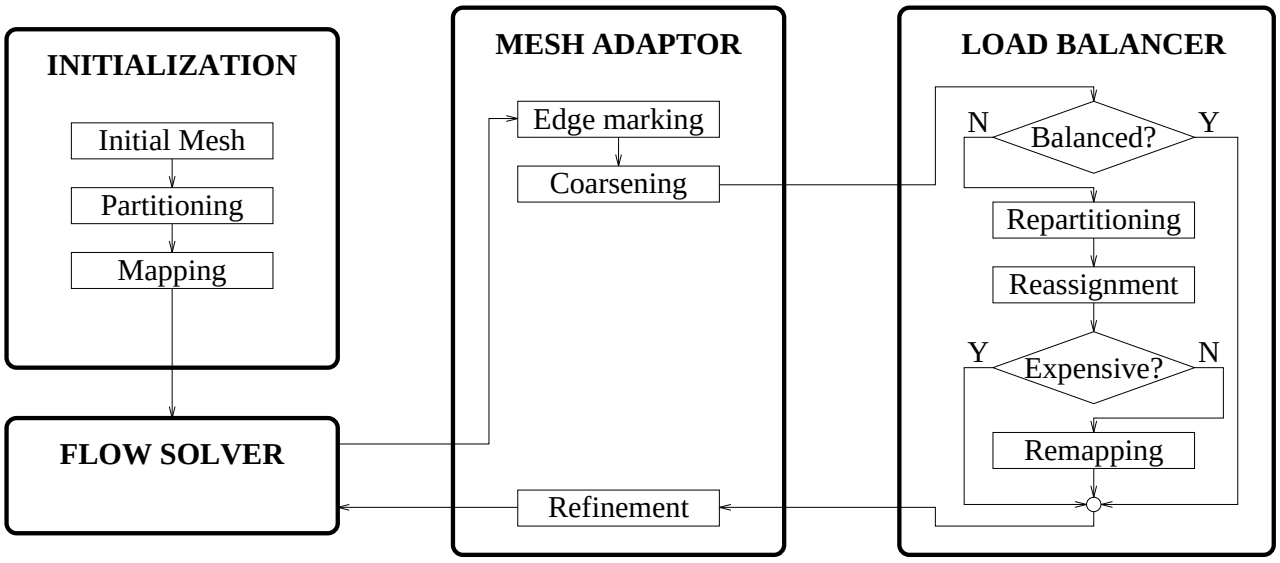


Figure 1: Overview of PLUM, our framework for parallel adaptive flow computation.

Our goal is to build a portable system for efficiently performing adaptive large-scale flow calculations in a parallel message-passing environment. Figure 1 depicts our framework, called PLUM, for such an automatic system. It consists of a flow solver and a mesh adaptor, with a partitioner and a remapper that load balances and redistributes the computational mesh when necessary. 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. It first targets edges for coarsening and refinement based on an error indicator computed from the flow 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 flow calculation is restarted.

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. However, the primary task of the load balancer is to balance the computational load for the flow solver while reducing the runtime communication. This is important because flow solvers are usually several times more expensive than mesh adaptors. In any case, it is obvious 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.

2. EULER FLOW SOLVER

An important component of PLUM is a numerical solver. Since we are currently interested in rotorcraft computational fluid dynamics (CFD) problems, we have chosen an unstructured-grid Euler flow solver [30] for the numerical calculations in this paper. It is a finite-volume upwind code that solves for the unknowns at the vertices of the mesh and satisfies the integral conservation laws on nonoverlapping polyhedral control volumes surrounding these vertices. Improved accuracy is achieved by using a piecewise linear reconstruction of the solution in each control volume. For helicopter problems, the Euler equations are written in an inertial reference frame so that the rotor blade and grid move through stationary air at the specified rotational and translational speeds. Fluxes across each control volume are computed using the relative velocities between the moving grid and the stationary far field. For a rotor in hover, the grid encompasses an appropriate fraction of the rotor azimuth. Periodicity is enforced by forming control volumes that include information from opposite sides of the grid domain. The solution is advanced in time using conventional explicit procedures.

The code uses an edge-based data structure that makes it particularly compatible with the mesh adaption procedure that we have incorporated within PLUM. Furthermore, since the number of edges in a mesh is significantly smaller than the number of faces, cell-vertex edge schemes are inherently more efficient than cell-centered element methods. Finally, an edge-based data structure does not limit the user to a particular type of volume element. Even though tetrahedral elements are used in this paper, any arbitrary combination of polyhedra can be used [6]. This is also true for our dynamic load balancing procedure.

3. PARALLEL MESH ADAPTION

A significant amount of research has been done to design sequential algorithms to effectively use unstructured meshes for the solution of fluid flow applications. Unfortunately,

many of these techniques cannot take advantage of the power of parallel computing due to the difficulties of porting these codes onto distributed-memory architectures. Recently, several two-dimensional adaptive methods have been successfully developed in a parallel environment, and some progress has been made towards three-dimensional adaptive unstructured-mesh schemes [21,24,26,27].

We have chosen the 3D_TAG scheme as the three-dimensional unstructured mesh adaption procedure within PLUM. The serial algorithm is extensively described in [4,5]. The 5000-line C code has its data structures based on edges of a tetrahedral mesh. This means that the elements are defined by their six edges rather than by their four vertices. This feature makes 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. 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. When an edge is bisected, the solution vector is linearly interpolated at the mid-point from the two points that constitute the original edge.

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 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 then 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. 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 parents are then subdivided based on their new patterns by invoking the mesh refinement procedure. As a result, the coarsening and refinement procedures share much of the same logic.

Details of the distributed-memory implementation are given in [24]. The parallel version consists of an additional 3000 lines of C++ and MPI code as a wrapper around the original serial mesh adaption program. An object-oriented approach allowed this to be performed in a clean and efficient manner. The parallel 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$\leftrightarrow$load-balancing cycle within PLUM shown in Fig. 1. The execution step runs a local copy of the mesh adaption algorithm 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.

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. Communication may be required to upgrade elements to one of the three allowed subdivision patterns. However, 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.

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. 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.

4. DYNAMIC LOAD BALANCING

In this paper, we present a novel method, called PLUM, to dynamically balance the processor workloads for unstructured adaptive-grid computations with a global view. Portions of this work reported earlier [2,3,23,29] have successfully demonstrated the viability and effectiveness of our load balancing framework. All major components within PLUM have now been completely implemented and integrated. This includes interfacing the parallel mesh adaption procedure based on actual flow solutions to a data remapping module, and incorporating an efficient parallel mesh repartitioner. A data remapping cost model is also proposed that can accurately predict the total cost of data redistribution 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.

4.1. Dual Graph of Initial Mesh

Parallel implementation of CFD flow solvers usually require a partitioning of the computational mesh, such that each tetrahedral element belongs to an unique partition. Communication is required across faces that are shared by adjacent elements residing on different

processors. Hence for the purposes of partitioning, we consider the dual of the computational mesh.

Using the dual graph representation of the *initial* mesh for the purpose of dynamic load balancing is one of the key features of this work. The tetrahedral elements of this 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. There is a significant advantage of using the dual of the initial computational mesh to perform the repartitioning and remapping at each load balancing step of PLUM. This is because the complexity remains unchanged during the course of an adaptive computation.

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. 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} 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 the weights w_{comp} and w_{remap} for every vertex and to the weight w_{comm} for every edge in the dual mesh. As a result, the repartitioning and load-balancing times depend only on the initial problem size and the number of partitions, but not on the size of the adapted mesh.

One minor disadvantage of using the initial dual grid is when the starting 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 usually be poor. One can then allow the initial mesh to be adapted one or more times before forming the dual graph that is then used for all future adaptions.

4.2. Preliminary Evaluation

Before embarking on an intensive load balancing phase, it is worthwhile estimating if the impending mesh adaption is going to seriously imbalance the processor workloads. The preliminary evaluation step achieves this goal by rapidly determining 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.

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 $(W_{\text{max}} - W_{\text{avg}})$. This is an exact measure of

the load imbalance. The mesh is repartitioned if the imbalance factor $W_{\max}/W_{\text{avg}}$ is unacceptable.

4.3. Parallel Mesh Repartitioning

If the preliminary evaluation step determines that the dual graph with a new weight distribution is unbalanced, the mesh needs to be repartitioned. Note that repartitioning is always performed on the initial dual graph with the weights of the vertices and edges adjusted to reflect a mesh adaption step. 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 PLUM load balancing framework to be viable. Serial partitioners are inherently inefficient since they do not scale in either time or space with the number of processors. Additionally, a bottleneck is created when all processors are required to send their portion of the grid to the host responsible for performing the partitioning. The solution must then be scattered back to all the processors before the load balancing can continue. A high quality parallel partitioner is therefore necessary to alleviate these problems.

Some excellent parallel partitioning algorithms are now available [18,27,28,35]; however, we need one that is extremely fast while giving good load balance and low edge cuts. For the test cases in this paper, an alpha version of parallel MeTiS [18] was used as the repartitioner. MeTiS is a multilevel algorithm which has been shown to quickly produce high quality partitions. It reduces the size of the graph by collapsing vertices and edges using a heavy edge matching scheme, applies a greedy graph growing algorithm for partitioning the coarsest graph, and then uncoarsens it back using a combination of boundary greedy and Kernighan-Lin refinement to construct a partitioning for the original graph. A key feature of parallel MeTiS is the utilization of graph coloring to parallelize both the coarsening and the uncoarsening phases. An additional benefit of the algorithm is the potential reduction in remapping cost since parallel MeTiS, unlike the serial version, uses the previous partition as the initial guess for the repartitioning. Results indicate that this partitioner can be effectively used inside PLUM; however, any other partitioning algorithm can also be used as long as it quickly delivers partitions that are reasonably balanced and require minimal communication.

4.4. Similarity Matrix Construction

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 rationale behind allowing multiple partitions per processor is that performing data mapping at a finer granularity reduces the volume of data movement at the expense of partitioning and processor reassignment times. However, the simpler scheme of setting F to unity suffices for most practical applications. Quantitative effects of varying F for our test cases are shown in Section 5.

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. Since the partitioning algorithm is run in parallel, each processor can simultaneously compute one row of the

Old Processors \ New Partitions	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 2: An example of a similarity matrix S for $P = 4$ and $F = 2$. Only the non-zero entries are shown.

matrix, based on the mapping between its current subdomain and the new partitioning. This information is then gathered by a single host processor that builds the complete similarity matrix, computes the new partition-to-processor mapping, and scatters the solution back to the processors. Note that these gather and scatter operations require a minuscule amount of time since only one row of the matrix ($P \times F$ integers) needs to be communicated to the host processor. A similarity matrix for $P = 4$ and $F = 2$ is shown in Fig. 2. Only the non-zero entries are shown.

4.5. Processor Reassignment

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. We present two general metrics: `TotalV` and `MaxV`, which model the remapping cost on most multiprocessor systems. `TotalV` minimizes the total volume of data moved among all processors, while `MaxV` minimizes the maximum flow of data to or from any single processor.

The metric `TotalV` assumes that by reducing network contention and the total number of elements moved, the remapping time will be reduced. In general, each processor cannot be assigned F unique partitions corresponding to their F largest weights. This is the case for the similarity matrix shown in Fig. 3(a) where the F largest weights for each processor are shaded. Note that $P = 4$ and $F = 1$ in this case. To minimize `TotalV`, each processor i must be assigned F unique partitions j_{i_f}, $f = 1, 2, \ldots, F$, so that the objective function

$$\mathcal{F} = \sum_{i=1}^{P} \sum_{f=1}^{F} S_{i,j_{i_f}}$$

is maximized subject to the constraint

$$j_{i_r} \neq j_{k_s}, \quad \forall i \neq k; \quad r, s = 1, 2, \ldots, F.$$

Both an optimal and a heuristic greedy algorithm have been implemented for solving this problem. When $F = 1$, the problem trivially reduces to a maximally weighted bipartite graph (MWBG), with P processors and P partitions in each set. An edge of weight $S_{i,j}$ 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 MWBG problem by duplicating each

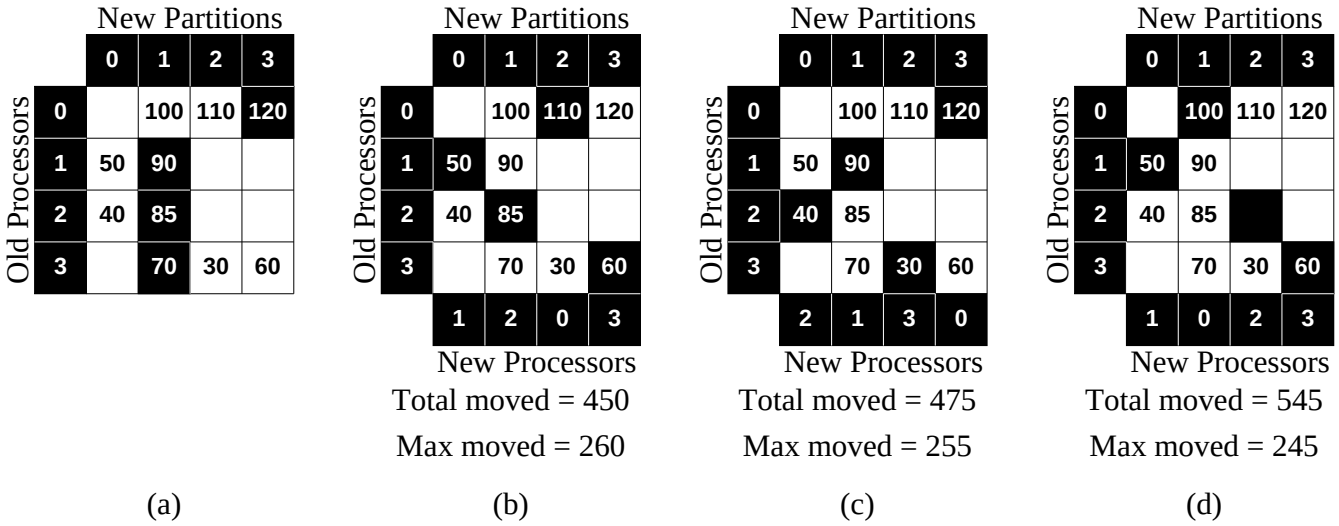


Figure 3: A similarity matrix S for $P = 4$ and $F = 1$ (a) before, and (b-d) after processor reassignment using (b) optimal MWBG algorithm and `TotalV` metric, (c) heuristic MWBG algorithm and `TotalV` metric, and (d) optimal BMCM algorithm and `MaxV` metric.

processor and all of its incident edges F times. Each set of the bipartite graph then has $P \times F$ vertices. After the optimal solution is obtained, the solutions for all F copies of a processor are combined to form a one-to-F mapping between the processors and the partitions. The optimal solution and the corresponding processor assignment using the `TotalV` metric for the similarity matrix in Fig. 3(a) is shown in Fig. 3(b). The optimal algorithm requires $O(VE)$ steps, where V and E are the number of vertices and edges in the weighted bipartite graph, respectively.

```
for (j=0; j<npart; j++) part_map[j] = unassigned;
for (i=0; i<nproc; i++) proc_unmap[i] = npart / nproc;
generate list L of entries in S in descending order using radix sort;
count = 0;
while (count < npart) {
  find next entry S[i][j] in L such that
       proc_unmap[i] > 0 and part_map[j] = unassigned;
  proc_unmap[i]--;
  part_map[j] = assigned;
  count++;
  map partition j to processor i;
}
```

Figure 4: Pseudocode for our heuristic algorithm for solving the processor reassignment problem.

We have developed a heuristic greedy algorithm that gives a suboptimal solution in $O(E)$ steps. The pseudocode for our heuristic algorithm is given in Fig. 4. Initially, all partitions are flagged as unassigned and each processor has a counter set to F that indicates the re-

maining number of partitions it needs. The non-zero entries of the similarity matrix S are then sorted in descending order. Starting from the largest entry, partitions are assigned to processors that have less than F partitions until done. If necessary, the zero entries in S are also used. Applying this heuristic algorithm to the similarity matrix in Fig. 3(a) generates the new processor assignment shown in Fig. 3(c). The value of the objective function $\mathcal{F}$ is 280 for the heuristic solution but is 305 for the optimal solution.

Theorem 1: *The value of the objective function $\mathcal{F}$ using the heuristic algorithm is always greater than half the optimal solution.*

Proof: We prove by the method of induction. Let $S^k_{i,j}$ denote the entry in the i-th row and j-th column of a $k \times k$ similarity matrix. Let $\texttt{Opt}^k$ and $\texttt{Heu}^k$ denote the optimal and heuristic solutions, respectively, for the similarity matrix S^k. When $k = 1$, $\texttt{Opt}^1 = \texttt{Heu}^1$ since there is only one entry in S^1 and must be chosen by both algorithms. Thus, $2\,\texttt{Heu}^1 \geq \texttt{Opt}^1$.

Assume now that the theorem is true for some $n \geq 1$; that is, $2\,\texttt{Heu}^n \geq \texttt{Opt}^n$. We need to show that $2\,\texttt{Heu}^{n+1} \geq \texttt{Opt}^{n+1}$.

Without loss of generality, create S^{n+1} from S^n by adding a new row and column such that $S^{n+1}_{n+1,n+1} \geq \max\,(S^{n+1}_{i,n+1},\, S^{n+1}_{n+1,i})$ for $1 \leq i \leq n$. Therefore, by definition of the heuristic algorithm, $\texttt{Heu}^{n+1} = \texttt{Heu}^n + S^{n+1}_{n+1,n+1}$. Since $2\,\texttt{Heu}^n \geq \texttt{Opt}^n$, we get $2\,\texttt{Heu}^{n+1} \geq \texttt{Opt}^n + 2\,S^{n+1}_{n+1,n+1}$. There are now two cases that can occur for the optimal solution.

Case 1. $S^{n+1}_{n+1,n+1}$ is contained in the optimal solution.
This means $\texttt{Opt}^{n+1} = \texttt{Opt}^n + S^{n+1}_{n+1,n+1}$. Thus, $2\,\texttt{Heu}^{n+1} \geq \texttt{Opt}^{n+1} + S^{n+1}_{n+1,n+1}$, which implies $2\,\texttt{Heu}^{n+1} \geq \texttt{Opt}^{n+1}$. □

Case 2. $S^{n+1}_{n+1,n+1}$ is not contained in the optimal solution.
Without loss of generality, assume that $S^{n+1}_{n,n+1}$ and $S^{n+1}_{n+1,n}$ are contained in the optimal solution. This means $\texttt{Opt}^{n+1} = \texttt{Opt}^{n-1} + S^{n+1}_{n,n+1} + S^{n+1}_{n+1,n}$. By definition of $S^{n+1}_{n+1,n+1}$, we get $\texttt{Opt}^{n+1} \leq \texttt{Opt}^{n-1} + 2\,S^{n+1}_{n+1,n+1}$. Since $\texttt{Opt}^n \geq \texttt{Opt}^{n-1}$, we have $\texttt{Opt}^{n+1} \leq \texttt{Opt}^n + 2\,S^{n+1}_{n+1,n+1}$. Therefore, $2\,\texttt{Heu}^{n+1} \geq \texttt{Opt}^{n+1}$. □

Corollary: *A processor assignment obtained using the heuristic algorithm can never result in a data movement cost that is more than twice that of the optimal assignment.*

Proof: We assume that the data movement cost is proportional to the number of elements that are moved and is given by $\sum\sum S_{i,j} - \mathcal{F}$. We need to show that $\sum\sum S^n_{i,j} - \texttt{Heu}^n \leq 2\,(\sum\sum S^n_{i,j} - \texttt{Opt}^n)$; that is, $\sum\sum S^n_{i,j} - 2\,\texttt{Opt}^n + \texttt{Heu}^n \geq 0$.

Let $\texttt{Int}^k$ be the sum of the similarity matrix entries that are contained in both $\texttt{Opt}^k$ and $\texttt{Heu}^k$. Therefore, $\sum\sum S^n_{i,j} \geq \texttt{Opt}^n + \texttt{Heu}^n - \texttt{Int}^n$. This implies $\sum\sum S^n_{i,j} - 2\,\texttt{Opt}^n + \texttt{Heu}^n \geq 2\,\texttt{Heu}^n - \texttt{Opt}^n - \texttt{Int}^n$. By Theorem 1, $2\,(\texttt{Heu}^n - \texttt{Int}^n) \geq (\texttt{Opt}^n - \texttt{Int}^n)$, since $(\texttt{Heu}^n - \texttt{Int}^n)$ and $(\texttt{Opt}^n - \texttt{Int}^n)$ are the heuristic and optimal solutions for a similarity matrix $S^k \subseteq S^n$. □

The metric `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. During the process of remapping, each processor must pack and unpack send and receive buffers, incur remote-memory latency time, and perform the computational overhead of rebuilding internal and shared data structures. By minimizing $\alpha \times \max(\texttt{ElemsSent})$ and $\beta \times \max(\texttt{ElemsRecd})$ (where

α and β are machine-specific parameters), `MaxV` attempts to reduce the total remapping time by minimizing the execution time of the most heavily-loaded processor. This problem can be solved optimally as the bottleneck maximum cardinality matching (BMCM) problem [13] in $O((V \log V)^{1/2}E)$ steps, and has been implemented for $F = 1$. The new processor assignment for the similarity matrix in Fig. 3(a) using this approach with $\alpha = \beta = 1$ is shown in Fig. 3(d). Notice that the total number of elements moved in Fig. 3(d) is larger than the corresponding value in Fig. 3(b); however, the maximum number of elements moved is smaller.

Note that `TotalV` does not consider the execution times of bottleneck processors while `MaxV` ignores bandwidth contention. A quantitative comparison of the two metrics for our test cases is presented in Section 5. In general, the objective function may need to use a combination of both metrics to effectively incorporate all related costs.

4.6. Cost Calculation

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 $\alpha = \beta$.

A significant amount of work has been done to model communication overhead including LogP [9], LogGP [1], and BSP [32]. All three 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 (also known as a single port model). Note that these models do not account for network contention (hotspots), since they

are extremely difficult to capture. Finally, BSP and LogGP arrive at similar cost metrics for bulk collective communication. Our redistribution procedure closely follows the superstep model of BSP.

All reported results were performed on the wide-node IBM SP2 located at NASA Ames Research Center. The processors are connected through a high performance switch, call the Vulcan chip. Each chip connects up to eight processors, and eight Vulcan chips comprise a switching board. An advantage of this interconnection mechanism is that all nodes can be considered equidistant from one another. This allows us to predict the 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 properly redistribute all the data.

4.7. Data Remapping

The remapping phase is responsible for physically moving data when it is reassigned to a different processor. It is generally the most expensive phase of any load balancing strategy. This data movement time can be significantly reduced by considering two distinct phases of mesh refinement: marking and subdivision. During the marking phase, edges are chosen for bisection either based on an error indicator or due to the propagation needed for valid mesh connectivity [4]. This is essentially a bookkeeping step during which the grid remains unchanged. The subdivision phase is the process of actually bisecting edges and creating new vertices and elements based on the generated edge-marking patterns. During this phase, the data volume corresponding to the grid grows since new mesh objects are created.

A key observation is that data remapping for a refinement step should be performed after the marking phase but before the actual subdivision. Because the refinement patterns are determined during the marking phase, the weights of the dual graph can be adjusted as though subdivision has already taken place. Based on the updated dual graph, the load balancer proceeds in generating a new partitioning, computing the new processor assignments, and performing the remapping on the original unrefined grid. Since a smaller volume of data is moved using this technique, a potentially significant cost savings is achieved. The newly redistributed mesh is then subdivided based on the marking patterns. This is the strategy that is used in PLUM (cf. Fig. 1).

An additional performance benefit is obtained as a side effect of this strategy. Since the original mesh is redistributed so that mesh refinement creates approximately the same number of elements in each partition, the subdivision phase performs in a more load balanced fashion. This reduces the total mesh refinement time. The savings should thus be incorporated as an additional term in the computational gain expression described in the previous subsection. The new partitioning and mapping are accepted if the computational gain is larger than the redistribution cost:

$$T_{\text{iter}} N_{\text{adapt}} (W_{\text{max}}^{\text{old}} - W_{\text{max}}^{\text{new}}) + T_{\text{refine}} \left(\frac{W_{\text{max}}^{\text{new}}}{W_{\text{max}}^{\text{old}}} - 1 \right) > \gamma \times \texttt{MaxSR} + O,$$

where T_{refine} is the time required to perform the subdivision phase based on the edge-marking patterns.

5. RESULTS

PLUM has been implemented on the IBM SP2 distributed-memory multiprocessor located at NASA Ames Research Center. The code is written in 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 paper.

The computational mesh used for most of the experiments in this paper is the one used to simulate the acoustics wind-tunnel test of Purcell [25]. In that experiment, 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 [31]. A cut-out view of the initial tetrahedral mesh is shown in Fig. 5.

In the first set of experiments, only one level of adaption is performed with varying fractions of the mesh in Fig. 5 being targeted for refinement. Three different cases are studied. The strategies, called **Real_1**, **Real_2**, and **Real_3**, subdivided 5%, 33%, and 60% of the 78,343 edges of the initial computational mesh. Edges are targeted for subdivision based on an error indicator [23] calculated directly from the flow solution. Table I lists the grid sizes for this single level of refinement for each of the three cases.

Figure 6 illustrates the parallel speedup for each of the three edge-marking strategies. Two sets of results are presented: one when data remapping is performed after mesh refinement, and the other when remapping is performed before refinement. The **Real_3** case shows the best speedup performance because it is the most computation intensive. Remapping the data before refinement has the largest relative effect for **Real_1**, increasing the

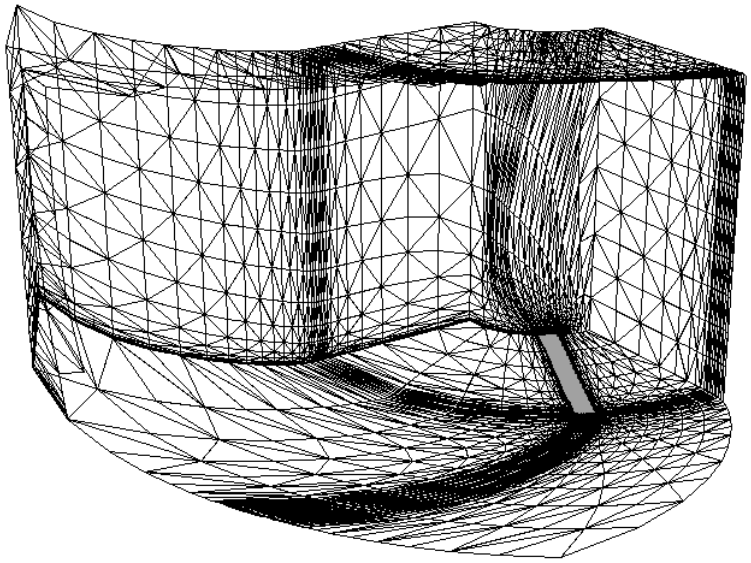


Figure 5: Cut-out view of the initial tetrahedral mesh.

TABLE I
Grid Sizes for the Three Different Refinement Strategies

	Vertices	Elements	Edges	Bdy Faces
Initial Mesh	13,967	60,968	78,343	6,818
Real_1	17,880	82,489	104,209	7,682
Real_2	39,332	201,780	247,115	12,008
Real_3	61,161	321,841	391,233	16,464

speedup from 9.3X to 23.9X on 64 processors. This is because the refinement region is the smallest for this strategy and load balancing the refined mesh before actual subdivision returns the biggest benefit. The results are the best for **Real_3** with data remapping before refinement, showing a 52.5X speedup on 64 processors. Extensive performance analysis of the parallel mesh adaption code is given in [24].

Figure 7 shows the remapping time for each of the three cases. As in Fig. 6, results are presented when the data remapping is done both after and before the actual mesh subdivision. A significant reduction in remapping time is observed when the adapted mesh is load balanced by performing data movement prior to actual subdivision. This is because the mesh grows in size only after the data has been redistributed. The biggest improvement is seen for **Real_3** when the remapping time is reduced to less than a third from 3.71 secs to 1.03 secs on 64 processors. These results in Figs. 6 and 7 demonstrate that our methodology within PLUM is effective in significantly reducing the data remapping time and improving the parallel performance of mesh refinement.

Figure 8 compares the execution times and the amount of data movement for the **Real_2** strategy when using the optimal and heuristic MWBG processor assignment algorithms. Both algorithms use the `TotalV` metric. Four pairs of curves are shown in each plot for $F = 1$, 2, 4, and 8. The optimal method always requires almost two orders of magnitude

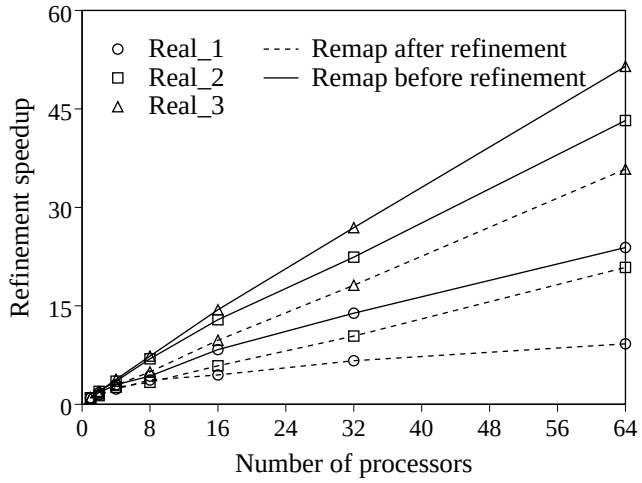


Figure 6: Speedup of the 3D_TAG parallel mesh adaption code when data is remapped either after or before mesh refinement.

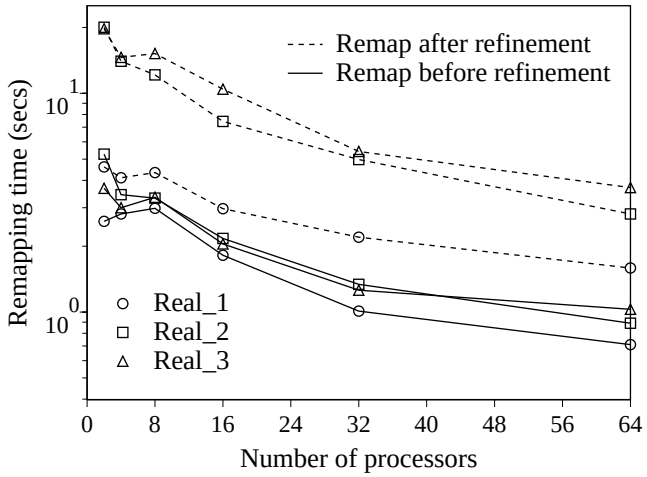


Figure 7: Remapping times within PLUM when data is remapped either after or before mesh refinement.

more time than our heuristic method. The execution times also increase significantly as F is increased 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. 8 illustrate that our heuristic mapper is almost as good as the optimal algorithm while requiring significantly less time. Similar results were obtained for the other edge-marking strategies.

Table II presents additional comparisons for the REAL_2 strategy in terms of the processor reassignment time and the amount of data movement. Since $F = 1$ in Table II, results for the optimal BMCM algorithm are also included. Unlike the MWBG algorithms which use the `TotalV` metric, the BMCM algorithm uses the `MaxV` metric. The optimal BMCM method always requires more time than the optimal MWBG method. The execution times for all three methods increase with the number of processors because of the growth in the size of

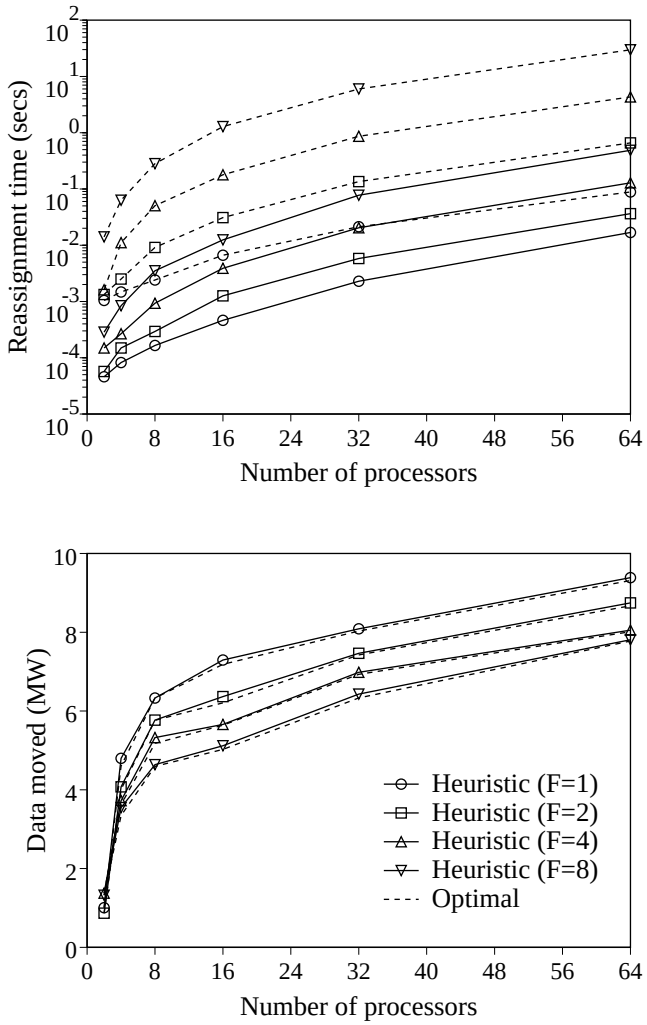


Figure 8: Comparison of the optimal and heuristic MWBG remappers in terms of the execution time (top) and the volume of data movement (bottom) for the REAL_2 strategy.

the similarity matrix; however, the heuristic MWBG time for 64 processors is still very small and acceptable. The total volume of data movement is obviously smaller for the MWBG algorithms because they use the `TotalV` cost metric. In the optimal BMCM method, the maximum of the number of elements sent or received is explicitly minimized; however, the MWBG methods give identical numbers. These values are shown in the second column of Table II. There were some differences in the maximum number of elements received among the three methods; however, the maximum number of elements sent was consistently larger and these are consequently reported. This demonstrates that for our test case, the heuristic algorithm does an excellent job of minimizing both the `TotalV` and the `MaxV` cost metrics. Similar results were obtained for the other two strategies. Although theoretical bounds have not been established, empirical evidence indicates that our heuristic MWBG algorithm closely approximates the `MaxV` solution. It was therefore used to perform the processor reassignment for all the experiments reported in this paper.

Figure 9 shows how the execution time is spent during the refinement and the subsequent

TABLE II
Comparison of the MWBG and BMCM Remappers for the **REAL_2** Strategy with $F = 1$

P	Max (Sent, Recd)	Opt MWBG Total Elems	Opt MWBG Reass. Time	Heu MWBG Total Elems	Heu MWBG Reass. Time	Opt BMCM Total Elems	Opt BMCM Reass. Time
2	11295	22522	0.0002	22522	0.0000	22522	0.0003
4	6827	16813	0.0004	16813	0.0001	16813	0.0006
8	8169	30071	0.0013	30071	0.0002	35506	0.0019
16	7131	35096	0.0045	36520	0.0005	50488	0.0070
32	4410	34738	0.0177	35032	0.0017	49641	0.0323
64	2264	38059	0.0650	38283	0.0088	52837	0.1327

load balancing phases for the three different cases. The reassignment times are not shown since they are negligible compared to the other times and are very similar to those listed in Table II for all the three cases. The repartitioning curves, using parallel MeTiS [18], are almost identical for the three cases because the time to repartition mostly depends on the initial problem size. Notice that the repartitioning times are almost independent of the number of processors; however, for our test mesh, there is a minimum when the number of processors is about 16. This is not unexpected. When there are too few processors, repartitioning takes more time because each processor has a bigger share of the total work. When there are too many processors, an increase in the communication cost slows down the repartitioner. For a larger initial mesh, the minimum partitioning time will occur for a higher number of processors. For **REAL_2**, the MeTiS partitioner required 0.58 secs to generate 64 partitions on 64 processors. 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. Notice that the refinement, repartitioning, and remapping times are generally comparable when using more than 32 processors. For example, the refinement and remapping phases required 0.55 secs and 0.89 secs, respectively, on 64 processors for **REAL_2**.

We also investigate the maximum and the actual impact of load balancing using PLUM on flow solver execution times. Suppose that P processors are used to solve a problem on a tetrahedral mesh consisting of N elements. In a load balanced configuration, each processor has N/P elements assigned to it. The computational mesh is then refined to generate a total of GN elements, $1 \leq G \leq 8$ for our refinement procedure. If the workload were balanced, each processor would have GN/P elements. But in the worst case, all the elements on a subset of processors are isotropically refined 1-to-8, while elements on the remaining processors remain unchanged. The most heavily-loaded processor would then have the smaller of $8N/P$ and GN–$(P$–$1)N/P$ elements. Thus, the maximum improvement due to load balancing for a *single* refinement step would be $\frac{1}{G}$ min $(8, P(G$–$1)+1)$.

The maximum impact of load balancing for the three strategies are shown in the top half of Fig. 10. The mesh growth factor G is 1.35 for the **REAL_1** case, giving a maximum improvement of 5.91 with load balancing when $P \geq 20$. The value of G is 3.31 and 5.28 for **REAL_2** and **REAL_3**, so the maximum improvements are 2.42 (for $P \geq 4$) and 1.52

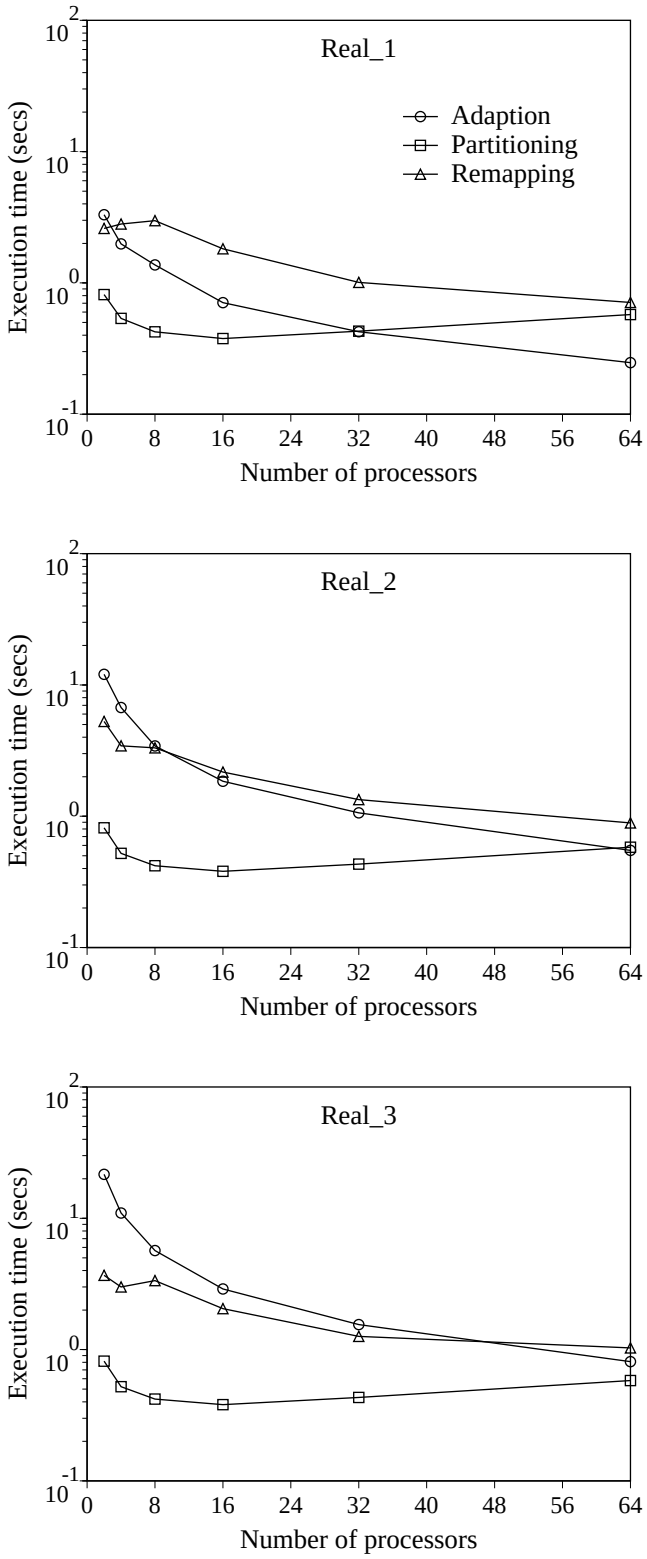


Figure 9: Anatomy of execution times for the REAL_1, REAL_2, and REAL_3 refinement strategies.

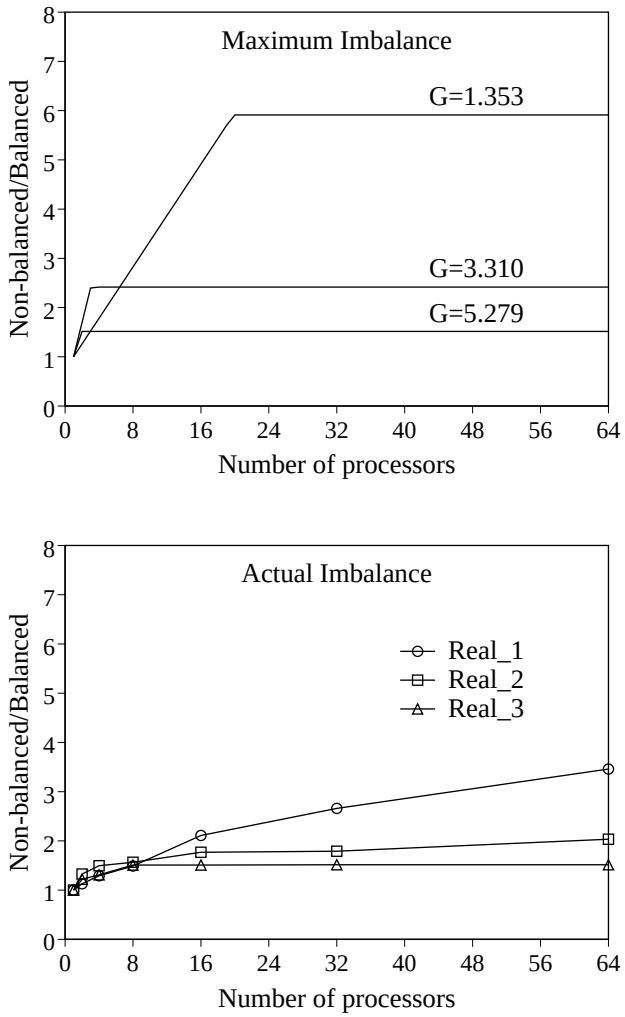


Figure 10: Maximum (top) and actual (bottom) impact of load balancing on flow solver execution times for different mesh growth factors G.

(for $P \geq 2$), respectively. There is obviously no improvement with load balancing if $G = 1$ or $G = 8$. Notice that maximum imbalance is attained faster as G increases; however, the magnitude of the maximum imbalance gradually decreases. The actual impact of load balancing is shown in the bottom half of Fig. 10. The three curves demonstrate the same basic nature as those for maximum imbalance. The improvement due to load balancing on 64 processors is a factor of 3.46, 2.03, and 1.52, for **Real_1**, **Real_2**, and **Real_3**, respectively. The impact of load balancing for these cases is somewhat less significant than the maximum possible since they model actual solution-based adaptions that do not necessarily cause worst case scenarios. Note, however, that the maximum improvement is already attained for **Real_3**. The **Real_1** and **Real_2** strategies would also attain their respective maxima if more processors were used. It is important to realize that the results shown in Fig. 10 are for a single refinement step. With repeated refinement, the gains realized with load balancing may be even more significant.

In the second set of experiments, a total of three levels of adaption are performed in

TABLE III
Progression of Grid Size through a Sequence of Three Levels of Adaption

	Vertices	Elements	Edges	Bdy Faces
Initial Mesh	13,967	60,968	78,343	6,818
Level 1	35,219	179,355	220,077	11,008
Level 2	72,123	389,947	469,607	15,076
Level 3	137,474	765,855	913,412	20,168

sequence on the mesh shown in Fig. 5. Table III shows the size of the computational mesh after each adaption step. Notice that the final mesh is more than an order of magnitude larger than the initial mesh. A close-up of the final mesh and pressure contours in the helicopter rotor plane are shown in Fig. 11. The mesh has been refined to adequately resolve the leading edge compression and capture both the surface shock and the resulting acoustic wave that propagates to the far field.

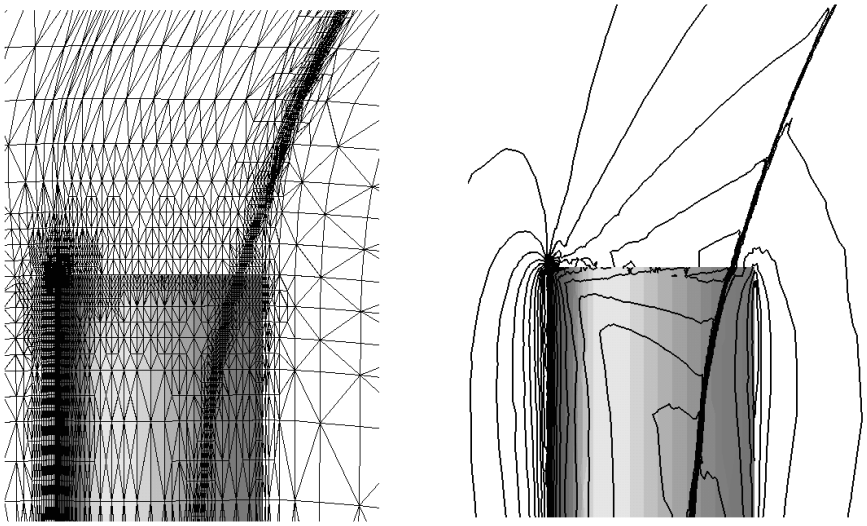


Figure 11: Final adapted mesh and computed pressure contours in the plane of the helicopter rotor.

Figure 12 shows how the execution time is spent during the adaption and the subsequent load balancing phases for the three levels. The reassignment times are not shown since they are several orders of magnitude smaller than the other times. The repartitioning curves, using parallel MeTiS [18], are almost identical to those shown in Fig. 9. Slight perturbations in the repartitioning times are due to different weight distributions of the dual graph. 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 time increases from one adaption level to the next because of the growth in the mesh size. 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

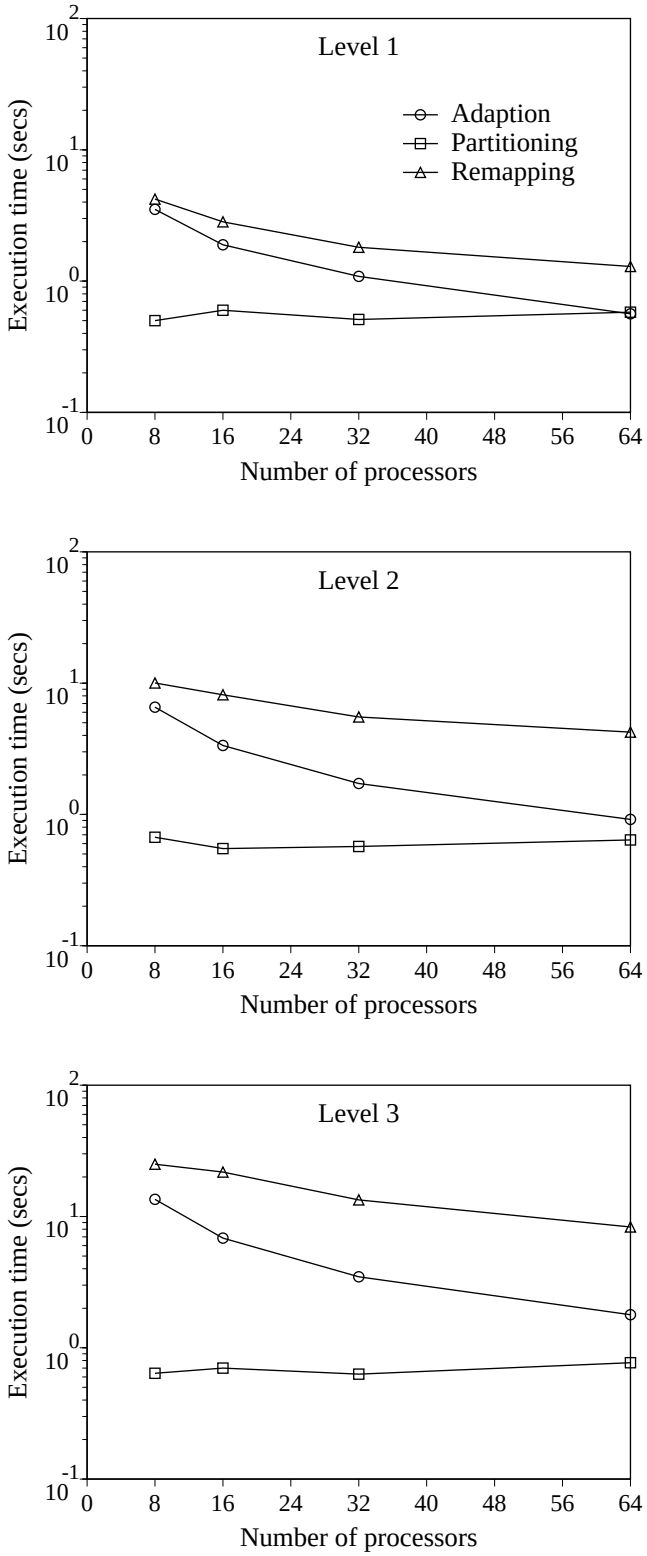


Figure 12: Anatomy of execution times for the three levels of adaption.

the bottleneck in dynamic load balancing problems. It is exactly for this reason that the remapping cost needs to be predicted accurately to be certain that the data redistribution cost will be more than compensated by the computational gain.

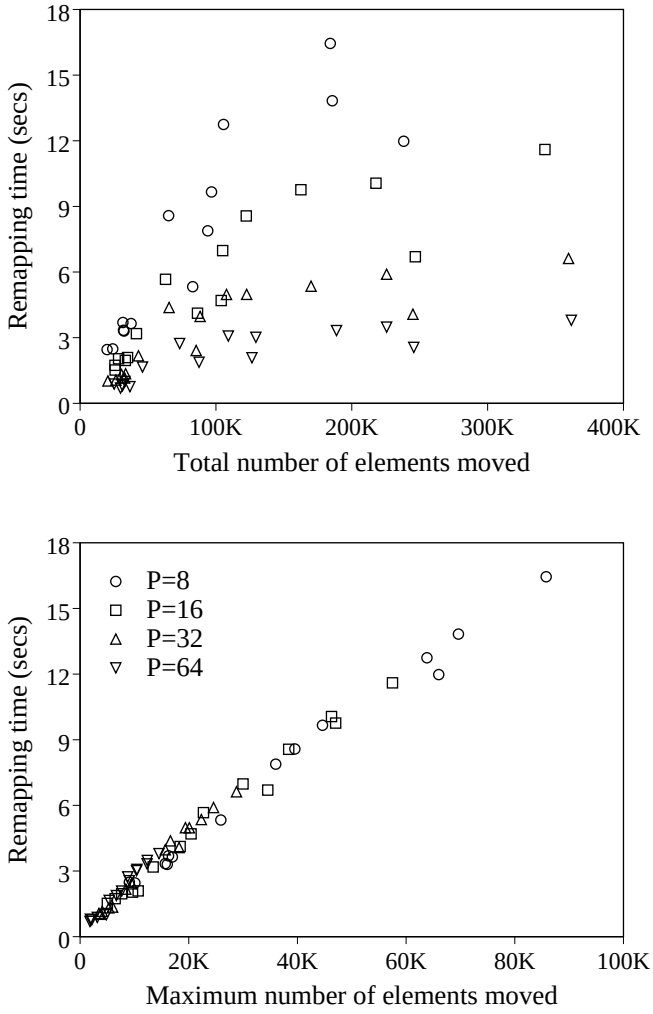


Figure 13: Remapping time as a function of the `TotalV` (top) and the `MaxSR` (bottom) metrics.

The third set of experiments are performed to compute the slope γ and the intercept O of our redistribution cost model. Empirical data is gathered by running various redistribution patterns. Data points are generated by permuting all possible combinations of the following four parameters: number of processors P (8,16,32,64), mesh growth factor G (1.4,3.3,5.3), remapping order (before refinement, after refinement), and similarity matrix solution (default, heuristic). This produces 48 redistribution times which are then plotted against two metrics, `TotalV` and `MaxSR`, in Fig. 13. Results demonstrate that there is little obvious correlation between the total number of elements moved (`TotalV` metric) 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` metric) 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 indicate 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.

6. CONCLUSIONS

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 PLUM, our framework for efficiently performing parallel adaptive numerical computations in a message-passing environment. It includes a novel method to dynamically balance the processor workloads with a global view. This paper presented the implementation and integration of all major components within PLUM. Several salient features of PLUM were described: (i) dual graph representation, (ii) parallel mesh repartitioner, (iii) optimal and heuristic remapping cost functions, (iv) efficient data movement and refinement schemes, and (v) accurate metrics comparing the computational gain and the redistribution cost. The code is written in C and C++ using the MPI message-passing paradigm and executed on an SP2.

Two different tests of PLUM were performed on a realistic-sized computational mesh used to simulate a helicopter acoustics experiment. The mesh adaption was based on numerical solutions obtained from an Euler flow solver. The first strategy targeted varying fractions of the initial tetrahedral mesh for refinement while the second strategy consisted of three successive levels of adaption. Results indicate that by using a high quality parallel partitioner to rebalance the work, a perfectly load balanced flow solver is guaranteed with minimum communication overhead.

We developed two generic metrics to model the remapping cost on most multiprocessor systems. Optimal solutions for both metrics, as well as a heuristic approach were implemented. It was shown that our heuristic algorithm quickly finds a solution which satisfies both metrics. Additionally, strong theoretical bounds on the heuristic time and solution quality were presented.

We also observed that data movement for a refinement step should be performed after the edge-marking phase but before the actual subdivision. This efficient remapping strategy resulted in almost a four-fold cost savings for data movement when 60% of the mesh was refined. A more load balanced refinement phase was an additional benefit of this approach. As a result, a three-fold improvement was observed in the refinement speedup.

Large-scale scientific computations on an SP2 showed that load balancing can dramatically reduce flow solver times over non-balanced loads. With multiple mesh adaptions, the gains realized with load balancing may be even more significant. Finally, a new remapping cost model for the SP2 was presented and quantitatively validated. Results indicated that reducing the bottleneck overhead guarantees a reduction in the total redistribution time.

In conclusion, we have shown that our parallel load balancing strategy for adaptive unstructured meshes will remain viable on large numbers of processors as none of the individual

modules will be a bottleneck.

REFERENCES

1. Alexandrov, A., Ionescu, M., Schauser, K. E., and Scheiman, C. LogGP: Incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. *Proc. 7th ACM Symposium on Parallel Algorithms and Architectures.* ACM SIGACT and SIGARCH, Santa Barbara, CA, 1995, pp. 95–105.

2. Biswas, R., and Oliker, L. Load balancing unstructured adaptive grids for CFD problems. *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing.* SIAM Activity Group on Supercomputing, Minneapolis, MN, 1997.

3. Biswas, R., Oliker, L., and Sohn, A. Global load balancing with parallel mesh adaption on distributed-memory systems. *Proc. Supercomputing '96.* ACM SIGARCH and IEEE Computer Society, Pittsburgh, PA, 1996.

4. Biswas, R., and Strawn, R. C. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Appl. Numer. Math.* **13,** 6 (February 1994), 437–452.

5. Biswas, R., and Strawn, R. C. Mesh quality control for multiply-refined tetrahedral grids. *Appl. Numer. Math.* **20,** 4 (April 1996), 337–348.

6. Biswas, R., and Strawn, R. C. Tetrahedral and hexahedral mesh adaptation for CFD problems. *Appl. Numer. Math.* to appear.

7. Chrisochoides, N. Multithreaded model for the dynamic load-balancing of parallel adaptive PDE computations. *Appl. Numer. Math.* **20,** 4 (April 1996), 321–336.

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

9. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. LogP: Towards a realistic model of parallel computation. *Proc. 4th ACM Symposium on Principles and Practice of Parallel Programming.* ACM SIGPLAN, San Diego, CA, 1993.

10. Cybenko, G. Dynamic load balancing for distributed-memory multiprocessors. *J. Parallel Distrib. Comput.* **7,** 2 (October 1989) 279–301.

11. Deng, Y., McCoy, R. A., Marr, R. B., and Peierls, R. F. An unconventional method for load balancing. *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing.* SIAM Activity Group on Supercomputing, San Francisco, CA, 1995, pp. 605–610.

12. Diniz, P., Plimpton, S., Hendrickson, B., and Leland, R. Parallel algorithms for dynamically partitioning unstructured grids. *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing.* SIAM Activity Group on Supercomputing, San Francisco, CA, 1995, pp. 615–620.

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

14. Galtier, J. Automatic partitioning techniques for solving partial differential equations on irregular adaptive meshes. *Proc. 10th ACM International Conference on Supercomputing.* ACM SIGARCH, Philadelphia, PA, 1996, pp. 157–164.

15. Ghosh, B., and Muthukrishnan, S. Dynamic load balancing in parallel and distributed networks by random matchings. *Proc. 6th ACM Symposium on Parallel Algorithms and Architectures.* ACM SIGACT and SIGARCH, Cape May, NJ, 1994, pp. 226–235.

16. Hegarty, D., Kechadi, M., and Dawson, K. Dynamic domain decomposition and load balancing for parallel simulations of long-chained molecules. *Proc. PARA95 Workshop on Applied Parallel Computing in Physics, Chemistry and Engineering Science.* Lyngby, Denmark, 1995, pp. 303–312.

17. Horton, G. A multi-level diffusion method for dynamic load balancing. *Par. Comput.* **19,** 2 (February 1993) 209–229.

18. Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. Department of Computer Science Tech. Rep. 95-035, University of Minnesota, Minneapolis, MN, 1995.

19. Kohring, G. A. Dynamic load balancing for parallelized particle simulations on MIMD computers. *Par. Comput.* **21,** 4 (April 1995) 683–693.

20. Mahapatra, N., and Dutt, S. Random seeking: A general, efficient, and informed randomized scheme for dynamic load balancing. *Proc. 10th International Parallel Processing Symposium.* ACM SIGARCH and IEEE Computer Society, Honolulu, HI, 1996, pp. 881–885.

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

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

23. Oliker, L., and Biswas, R. Efficient load balancing and data remapping for adaptive grid calculations. *Proc. 9th ACM Symposium on Parallel Algorithms and Architectures.* ACM SIGACT and SIGARCH, Newport, RI, 1997, to appear.

24. Oliker, L., Biswas, R., and Strawn, R. C. Parallel implementation of an adaptive scheme for 3d unstructured grids on the SP2. In Ferreira, A., Rolim, J., Saad, Y., and Yang, T. (Eds.). *Parallel Algorithms for Irregularly Structured Problems.* Springer-Verlag, Berlin, Germany, LNCS 1117, 1996, pp. 35–47.

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

26. Selwood, P. M., Verhoeven, N. A., Nash, J. M., Berzins, M., Weatherill, N. P., Dew, P. M., and Morgan, K. Parallel mesh generation and adaptivity: Partitioning and analysis. In Schiano, P., Ecer, A., Periaux, J., and Satofuka, N. (Eds.). *Parallel Computational Fluid Dynamics: Algorithms and Results Using Advanced Computers.* Elsevier Science, Amsterdam, The Netherlands, 1997, pp. 166–173.

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

28. Simon, H. D., Sohn, A., and Biswas, R. HARP: A fast spectral partitioner. *Proc. 9th ACM Symposium on Parallel Algorithms and Architectures.* ACM SIGACT and SIGARCH, Newport, RI, 1997, to appear.

29. Sohn, A., Biswas, R., and Simon, H. D. Impact of load balancing on unstructured adaptive grid computations for distributed-memory multiprocessors. *Proc. 8th IEEE Symposium on Parallel and Distributed Processing.* IEEE Computer Society, New Orleans, LA, 1996, pp. 26–33.

30. Strawn, R. C., and Barth, T. J. A finite-volume Euler solver for computing rotary-wing aerodynamics on unstructured meshes. *J. AHS* **38,** 2 (April 1993) 61–67.

31. Strawn, R. C., Biswas, R., and Garceau, M. Unstructured adaptive mesh computations of rotorcraft high-speed impulsive noise. *J. Aircraft* **32,** 4 (July-August 1995) 754–760.

32. Valiant, L. G. A bridging model for parallel computation. *Comm. of the ACM* **33,** 8 (August 1990) 103–111.

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

34. Vidwans, A., Kallinderis, Y., and Venkatakrishnan, V. Parallel dynamic load-balancing algorithm for three-dimensional adaptive unstructured grids. *AIAA J.* **32,** 3 (March 1994) 497–505.

35. Walshaw, C., Cross, M., and Everett, M. G. Dynamic load-balancing for parallel adaptive unstructured meshes. *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing.* SIAM Activity Group on Supercomputing, Minneapolis, MN, 1997.