Efficient Load Balancing and Data Remapping for Adaptive Grid Calculations

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

Rupak Biswas
MRJ Technology Solutions, NASA Ames Research Center, Moffett Field, CA 94035
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 to dynamically balance the processor workloads with a global view. This paper presents, for the first time, 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. Previous results indicated that mesh repartitioning and data remapping are potential bottlenecks for performing large-scale scientific calculations. We resolve these issues and demonstrate that our framework 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 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. 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 [5-9,11-14,16-18,24-26]; however, most of them either lack a global view of loads across processors or do not apply their techniques to realistic large-scale applications.

Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures, Newport, Rhode Island, June 22–25, 1997.

Figure 1 depicts our framework for parallel adaptive flow computation. 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. Our goal is to build a portable system for efficiently performing adaptive large-scale flow calculations in a parallel message-passing environment. The mesh is first partitioned and mapped among the available processors. The flow solver then runs for several iterations, updating solution variables. Once an acceptable solution is obtained, the 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 can lead to a potentially significant savings in the redistribution cost. The load balancer also balances 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.

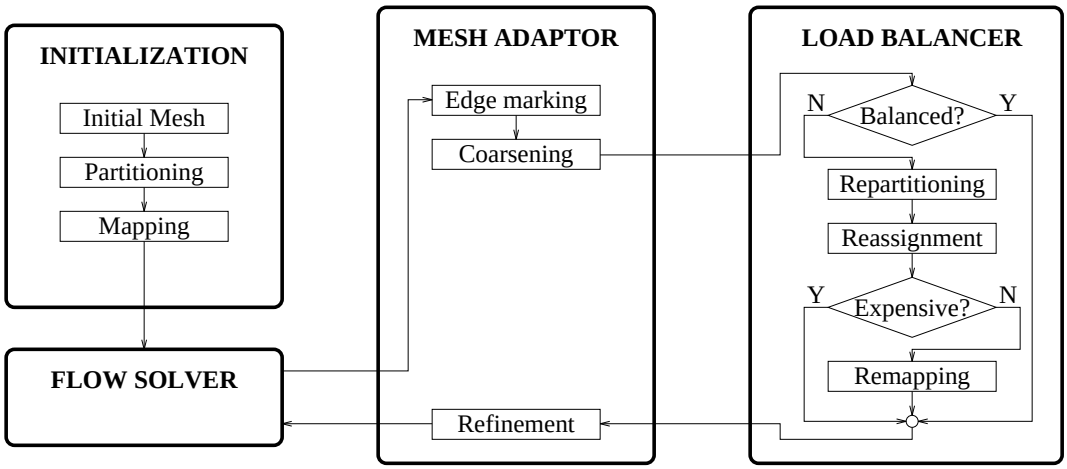


Figure 1: Overview of our framework for parallel adaptive flow computation.

2 EULER SOLVER FOR ROTOR FLOWS

The unstructured-grid CFD solver [22] used for the numerical calculations in this paper 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 our mesh adaption procedure. 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 [4]. This is also true for our load balancing procedure.

3 PARALLEL MESH ADAPTION

The serial mesh adaption scheme, called 3D_TAG, is described in [3]. 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.

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. Once this edge marking is completed, each element is independently subdivided based on its binary pattern. Special data structures are used to ensure that this process is computationally efficient.

Mesh coarsening also uses the edge-marking patterns. If a child element has any edge marked for coarsening, this element and its siblings are removed and their parent is reinstated. Parent edges and elements are retained at each refinement step so they do not have to be reconstructed. Reinstated parent elements have their edge-marking patterns adjusted to reflect that some edges have been coarsened. The 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.

There are some constraints for mesh coarsening. For example, edges cannot be coarsened beyond the initial mesh. Edges must also be coarsened in an order that is reversed from the one by which they were refined. Moreover, an edge can coarsen if and only if its sibling is also targeted for coarsening. More details about these coarsening constraints are given in [4].

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.

Details of the distributed-memory implementation are given in [19]. 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↔adaption↔load-balancing cycle 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. Shared vertices and edges are identified by searching for elements that lie on partition boundaries. A bit flag is set to distinguish between shared and internal objects. A list of shared processors (SPL) is also generated for each shared object. The additional storage that is required for the parallel code depends on the number of processors used and the fraction of shared objects. For the cases in this paper, this was less than 10% of the memory requirements of the serial version.

The execution phase runs a copy of 3D_TAG on each processor that adapts its local region, while maintaining a globally-consistent grid along partition boundaries. The first step is to target edges for refinement or coarsening based on an error indicator computed from the flow solution. This process results in a symmetrical marking of all shared edges across partitions because shared edges have the same flow and geometry information regardless of their processor number. However, elements have to be continuously upgraded to one of the three allowed subdivision patterns. This causes some propagation of edges targeted for refinement that could mark local copies of shared edges inconsistently. This is because the local geometry and marking patterns affect the nature of the propagation. Communication is therefore required after each iteration of the propagation process. Every processor sends a list of all the newly-marked local copies of shared edges to all the other processors in their SPLs. The process may continue for several iterations, and edge markings could propagate back and forth across partitions. Once all edge markings are complete, each processor executes the mesh adaption code without the need for further communication, since all edges are consistently marked. The only task remaining is to update the shared edge and vertex information as the mesh is adapted. This is handled as a post-processing phase.

New edges and vertices that are created on partition boundaries during refinement are assigned shared processor information. If a shared edge is bisected, its two children and the center vertex inherit its SPL. However, if a new edge is created that lies across an element face, communication is sometimes required to determine whether it is shared or internal. If it is shared, the SPL must be formed.

The coarsening phase purges the data structures of all edges that are removed, as well as their associated vertices, elements, and boundary faces. No new shared information is generated since no mesh objects are created during this step. However, objects are renumbered due to compaction and all internal and shared data are updated accordingly. The refinement routine is then invoked to generate a valid mesh from the vertices left after the coarsening.

It is sometimes necessary to create a single global mesh after one or more adaption steps. Some post processing tasks, such as visualization, need to 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

We present a novel method to dynamically balance the processor workloads with a global view. Results reported in [21] focused on fundamental load balancing issues while simulating various modules of our framework. This paper presents, for the first time, the implementation and integration of all major components within our dynamic load balancing strategy. This includes interfacing the parallel mesh adaption procedure based on actual flow solutions to a data remapping module. Previous results indicated that mesh repartitioning and data remapping are potential bottlenecks for performing large-scale flow analysis. We resolve these issues and demonstrate that our framework remains viable on a large number of 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

The dual graph representation of the initial computational mesh 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. The most significant advantage of using the dual of the initial computational mesh to perform the partitioning and mapping is that 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 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. New grids obtained by adaption are translated to the two weights for every element in the initial mesh. As a result, the repartitioning time depends 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 dual grid is when the initial computational mesh is either too large or too small. For extremely large initial meshes, the partitioning time will be excessive. This problem can be circumvented by agglomerating groups of elements into larger superelements. For very small meshes, the quality of the partitions will be bad. One can then allow the initial mesh to be adapted one or more times before using the dual graph for all future adaptions.

4.2 PARALLEL MESH REPARTITIONING

If the preliminary evaluation step determines that the dual graph with a new set of computational weights 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. The repartitioning phase must be performed very rapidly for our 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.

For the test cases in this paper, an alpha version of parallel MeTiS [15] was used for repartitioning. 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 partitioning algorithm can be effectively used inside our load balancing scheme. However, any partitioning algorithm could be used, as long as it is fast, and delivers reasonably balanced partitions based on the new weights.

4.3 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 and was used for the tests in this paper. The effects of varying F can be found in [2].

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 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 $F = 1$ is shown in Fig. 2. Only the non-zero entries are shown.

4.4 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.

`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. 2(a) where the F largest weights for each processor are shaded. 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 $j_{i_r} \neq j_{k_s}$, for all $i \neq k$ and $r, s = 1, 2, \ldots, F$.

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 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. 2(a) is shown in Fig. 2(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. We have developed a heuristic greedy algorithm that gives a suboptimal solution in $O(E)$ steps. The pseudocode

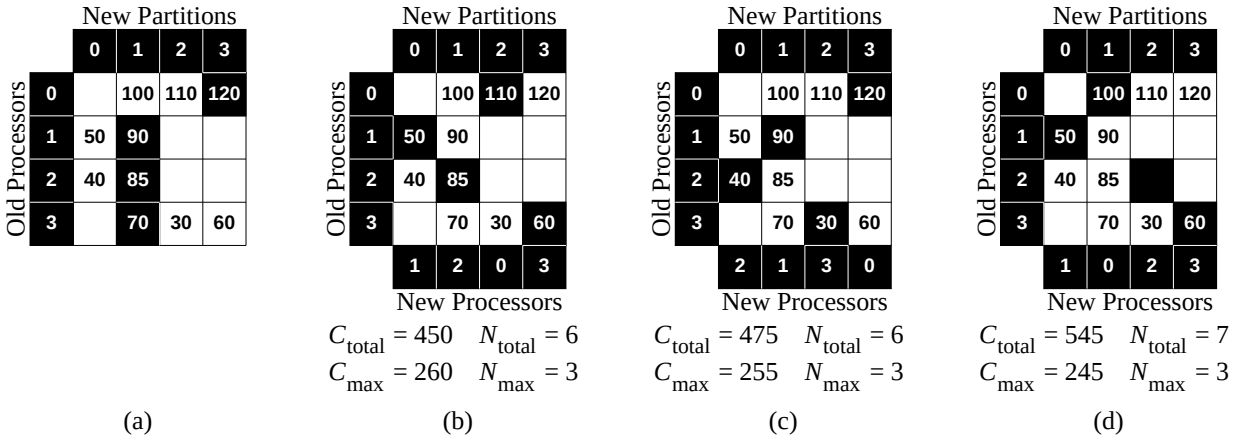


Figure 2: A similarity matrix (a) before, and (b-d) after processor reassignment using (b) optimal MWBG algorithm and `TotalV` metric, (c) heuristic MWBG algorithm and `TotalV` metric, and (d) optimal BMCM algorithm and `MaxV` metric.

for our heuristic algorithm is as follows:

```
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;
}
```

Initially, all partitions are flagged as unassigned and each processor has a counter set to F that indicates the remaining 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. 2(a) generates the new processor assignment shown in Fig. 2(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 $S^k \subseteq S^n$. □

`MaxV` 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 the maximum of $\alpha \times$#ElementsSent and $\beta \times$#ElementsReceived (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 [10] 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. 2(a) using this approach with $\alpha = \beta = 1$ is shown in Fig. 2(d).

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. This issue will be addressed in future work.

4.5 COST CALCULATION

The computational gain due to repartitioning is proportional to the decrease in the load imbalance achieved by running the adapted mesh on the new partitions rather than on the old partitions. 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 redistribution cost is calculated from the similarity matrix using two machine-dependent parameters: the remote-memory latency time T_{lat} and the message startup time T_{setup}. T_{lat} is the time required for memory-to-memory copying of a word, and applies to every initial mesh element that is moved. T_{setup} is the time required to prepare message headers, load the message buffer, and so on, and applies to each set of elements that is moved from one processor to another. For the `TotalV` metric, the redistribution cost can be expressed as $MC_{\text{total}}T_{\text{lat}} + N_{\text{total}}T_{\text{setup}}$, where M is the storage requirements per element for the solver and mesh adaptor, and C_{total} and N_{total} are the total number of elements and sets of elements to be moved, respectively. For the `MaxV` metric, the redistribution cost can be written as $MC_{\max}T_{\text{lat}} + N_{\max}T_{\text{setup}}$, where $C_{\max}$ and $N_{\max}$ are the total number elements and sets of elements to be moved for the bottleneck processor only. The values of C_{total}, $C_{\max}$, N_{total}, and $N_{\max}$ for each of the three processor reassignments are shown in Figs. 2(b)-2(d). The new partitioning and processor reassignment are accepted if the computational gain is larger than the redistribution cost.

4.6 EFFICIENT DATA REMAPPING

The remapping phase is responsible for physically moving data when it is reassigned to a different processor. When an element is moved from one processor to another, a communication cost as well as a computational overhead are incurred. The communication cost includes the time required to pack and unpack the send and receive buffers, and the message startup and remote-memory latency times. The computational overhead is the time necessary to rebuild the internal and shared data structures.

Previous results [1] have indicated that remapping is the most expensive phase of our load balancing strategy. This 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 [3]. 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 strategy, a potentially significant cost savings is achieved. The newly redistributed mesh is then subdivided based on the marking patterns.

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:

$$T_{\text{iter}} N_{\text{adapt}} (W_{\max}^{\text{old}} - W_{\max}^{\text{new}}) + T_{\text{refine}} \left(\frac{W_{\max}^{\text{new}}}{W_{\max}^{\text{old}}} - 1 \right) > MCT_{\text{lat}} + NT_{\text{setup}},$$

where T_{refine} is the time required to perform the subdivision phase based on the edge-marking patterns. The symbols C and N are C_{total} and N_{total} for the `TotalV` metric, and $C_{\max}$ and $N_{\max}$ for the `MaxV` metric.

5 RESULTS

The parallel mesh adaption and global load balancing procedures have been implemented on an IBM SP2. Both codes are written in C and C++, with the parallel activities in MPI for portability. No SP2-specific optimizations were used to obtain the results reported here.

The computational mesh is the one used to simulate an acoustics experiment of Purcell [20] where a 1/7th-scale model of a UH-1H helicopter rotor blade was tested over a range of subsonic and transonic hover-tip Mach numbers. Numerical results and a detailed report of the simulation are given in [23]. A cut-out view of the initial tetrahedral mesh is shown in Fig. 3.

Results are presented for one refinement step where edges are targeted for subdivision based on an error indicator [23] calculated directly from the flow solution. Three different cases are studied with varying fractions of the domain being targeted for refinement. The three strategies, called `Real_1`, `Real_2`, and `Real_3`, subdivided 5%, 33%, and 60% of the 78,343 edges of the initial computational mesh. Table 1 lists the grid sizes for this single level of refinement for each of the three cases. Coarsening results are not reported here as they are similar to those presented earlier [1,2]. We focus only on the mesh refinement phase as it is being done in a load balanced fashion for the first time while also reducing data movement. Several other edge-marking strategies based on geometry have been investigated elsewhere [1].

Figure 4 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 be-

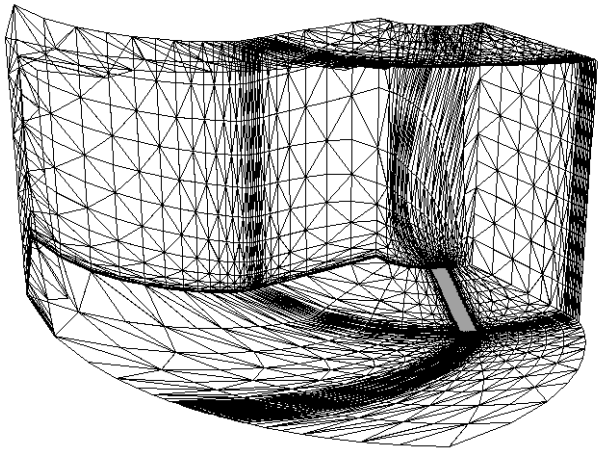


Figure 3: View of the initial tetrahedral mesh.

	Vertices	Elements	Edges	Bdy Faces
Initial	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

Table 1: Grid sizes for the three refinement strategies.

fore 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 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 [19].

Figure 5 shows the remapping time for each of the three cases. As in Fig. 4, results are presented both when the data remapping is done 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. 4 and 5 demonstrate that our methodology is effective in significantly reducing the data remapping time and improving the parallel performance of mesh refinement.

Table 2 compares the processor reassignment times and the amount of data movement for the `Real_2` strategy when using the optimal and the heuristic MWBG, and the optimal BMCM algorithms. The MWBG algorithms use the `TotalV` metric, whereas the BMCM algorithm uses the `MaxV` metric. The optimal MWBG method always requires almost an order of magnitude more time than the heuristic method; however, the reduction in the amount of total data movement is insignificant for our test case. The optimal BMCM method always requires more time than the optimal MWBG method. The execution times for all three methods increase

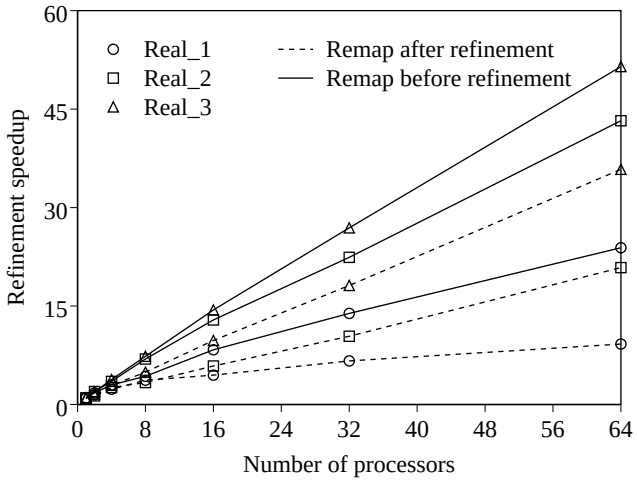


Figure 4: Speedup of the parallel mesh adaptor when data is remapped either after or before mesh refinement.

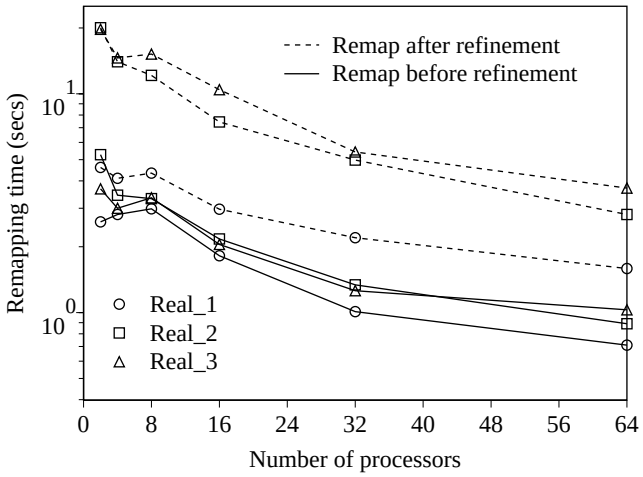


Figure 5: Remapping times when data is remapped either after or before mesh refinement.

with the number of processors because of the growth in the size of 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 2. 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.

Figure 6 shows how the execution time is spent during the refinement and the subsequent load balancing phases for the three different cases. The `TotalV` metric was used to model the remapping cost. Our heuristic greedy MWBG algorithm was used to perform the processor reassignment.

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

Table 2: Comparison of the optimal MWBG, heuristic MWBG (both using the `TotalV` cost metric), and optimal BMCM (using the `MaxV` cost metric) mappers in terms of the number of elements moved and the reassignment times (in secs) for the `Real_2` refinement strategy.

The reassignment times are not shown since they are negligible compared to the other times and are very similar to those listed in Table 2 for all the three cases. The repartitioning curves, using parallel MeTiS [15], 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 `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`. The results indicate that our load balancing strategy will remain viable on large numbers of processors as none of the individual modules will be a bottleneck.

Finally, we investigate the impact of load balancing 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 Fig. 7. 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 (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 maximum imbalance value gradually decreases.

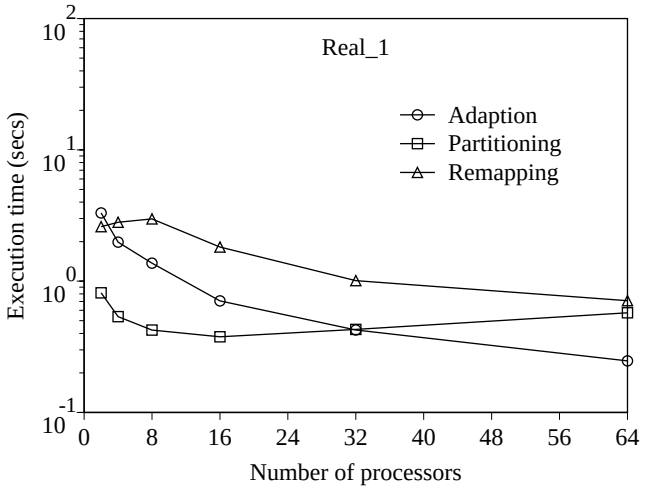
Figure 8 illustrates the actual impact of load balancing for the different cases. The three curves demonstrate the same basic nature as those in Fig. 7. The improvement due to load balancing on 64 processors is a factor of 3.46, 2.03, and 1.52, for `Real_1`, `Real_2`, and `Real_3`, respectively. The 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 Figs. 7 and 8 are for a single refinement step. With repeated refinement, the gains realized with load balancing may be even more significant.

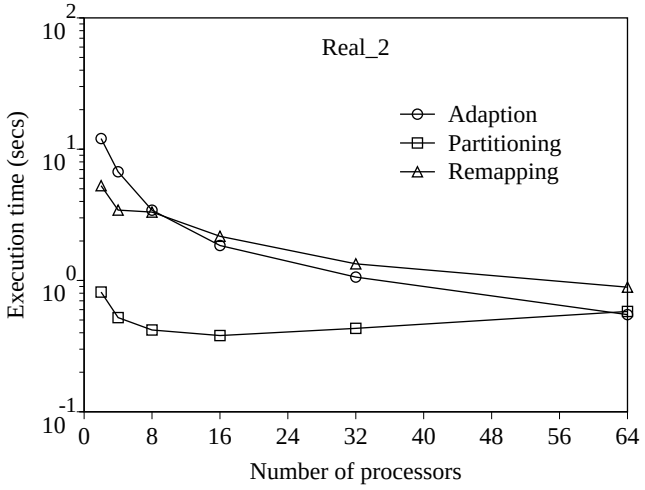
6 SUMMARY

We have described our framework for efficiently performing parallel adaptive flow computations in a message-passing environment. A novel method was presented to dynamically balance the processor workloads with a global view. This paper presented the implementation and integration of all major components within our load balancing strategy including the interface between a parallel mesh adaption code and a data remapping module. Several novel features of our framework 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.

Three different cases were studied with varying fractions of a realistic-sized domain being targeted for refinement. The mesh adaption was based on actual flow solutions for a helicopter rotor blade acoustics simulation. The three strategies subdivided an initial tetrahedral mesh of 60,968 elements into approximately 82,500, 201,800, and 321,800 elements. 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.





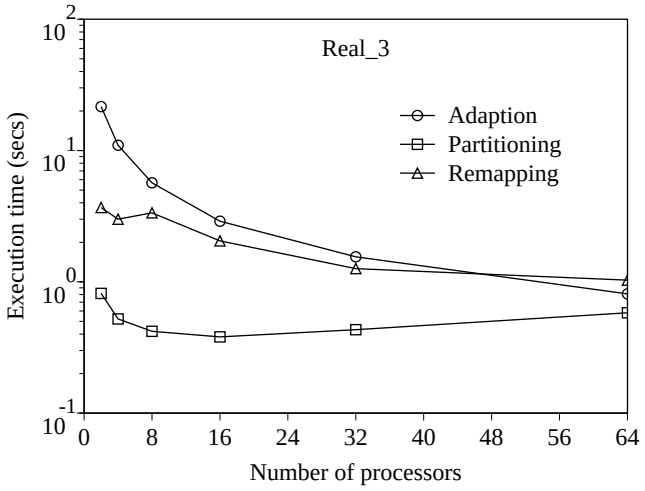


Figure 6: Anatomy of execution times for the three different refinement strategies. The refinement, partitioning, and remapping times for 64 processors are (0.25,0.57,0.71), (0.55,0.58,0.89), and (0.81,0.60,1.03) for `Real_1`, `Real_2`, and `Real_3`, respectively.

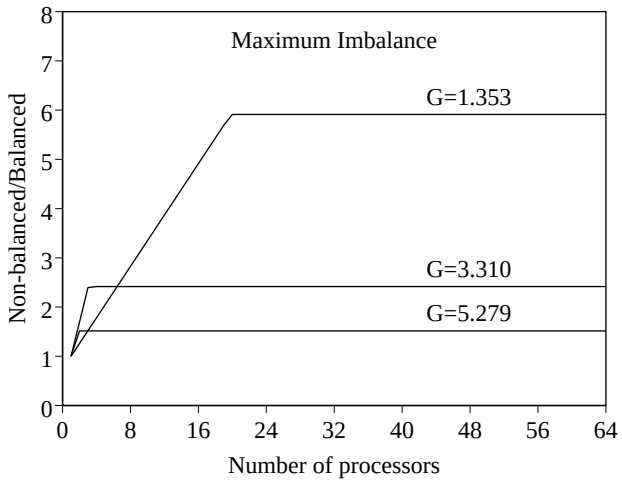


Figure 7: Maximum impact of load balancing on flow solver execution times for different mesh growth factors.

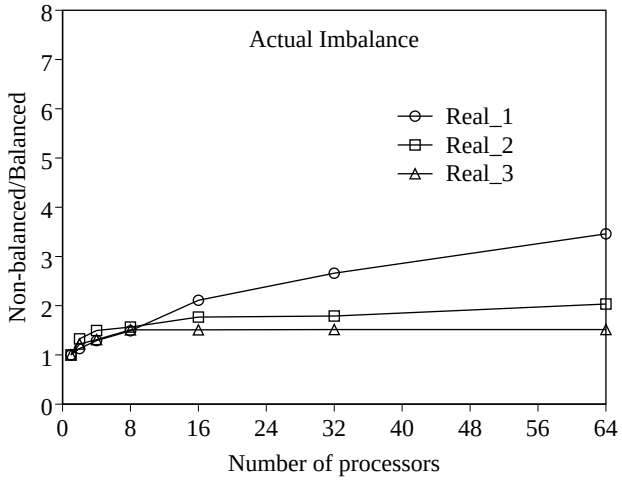


Figure 8: Actual impact of load balancing on flow solver execution times.

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 on the largest test case. A load balanced refinement phase was an additional benefit of this approach. As a result, an improvement of up to 2.6X was observed in the refinement speedup.

Finally, 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. Overall, the results have demonstrated that our framework will remain viable on a large number of processors.

7 ACKNOWLEDGEMENTS

The authors would like to thank Hal Gabow for observing that our processor assignment problem using the `MaxV` met-

ric is equivalent to a bottleneck maximum cardinality matching. The work of the first author was supported by NASA under Contract Number NAS 2-96027 with the Universities Space Research Association. The work of the second author was supported by NASA under Contract Number NAS 2-14303 with MRJ Technology Solutions.

REFERENCES

[1] R. Biswas and L. Oliker, "Load Balancing Unstructured Adaptive Grids for CFD Problems," *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, 1997.

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

[4] R. Biswas and R.C. Strawn, "Tetrahedral and Hexahedral Mesh Adaptation for CFD Problems," *Applied Numerical Mathematics*, to appear.

[5] N. Chrisochoides, "Multithreaded Model for the Dynamic Load-Balancing of Parallel Adaptive PDE Computations," *Applied Numerical Mathematics* **20** (1996) 349–365.

[6] H.L. de Cougny, K.D. Devine, J.E. Flaherty, R.M. Loy, C. Ozturan, and M.S. Shephard, "Load Balancing for the Parallel Adaptive Solution of Partial Differential Equations," *Applied Numerical Mathematics* **16** (1994) 157–182.

[7] G. Cybenko, "Dynamic Load Balancing for Distributed-Memory Multiprocessors," *J. Parallel and Distributed Computing* **7** (1989) 279–301.

[8] Y. Deng, R.A. McCoy, R.B. Marr, and R.F. Peierls, "An Unconventional Method for Load Balancing," *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, 1995, pp. 605–610.

[9] P. Diniz, S. Plimpton, B. Hendrickson, R. Leland, "Parallel Algorithms for Dynamically Partitioning Unstructured Grids," *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, 1995, pp. 615–620.

[10] H.N. Gabow and R.E. Tarjan, "Algorithms for Two Bottleneck Optimization Problems," *J. Algorithms* **9** (1988) 411–417.

[11] J. Galtier, "Automatic Partitioning Techniques for Solving Partial Differential Equations on Irregular Adaptive Meshes," *Proc. 10th ACM International Conference on Supercomputing*, Philadelphia, PA, 1996, pp. 157–164.

[12] B. Ghosh and S. Muthukrishnan, "Dynamic Load Balancing in Parallel and Distributed Networks by Random Matchings," *Proc. 6th ACM Symposium on Parallel Algorithms and Architectures* Cape May, NJ, 1994, pp. 226–235.

[13] D. Hegarty, M. Kechadi, and K. Dawson, "Dynamic Domain Decomposition and Load Balancing for Parallel Simulations of Long-Chained Molecules," *PARA95 Workshop on Applied Parallel Computing in Physics, Chemistry and Engineering Science*, Lyngby, Denmark, 1995, pp. 303–312.

[14] G. Horton, "A Multilevel Diffusion Method for Dynamic Load Balancing," *Parallel Computing* **19** (1993) 209–229.

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

[16] G. Kohring, "Dynamic Load Balancing for Parallelized Particle Simulations on MIMD Computers," *Parallel Computing* **21** (1995) 683–693.

[17] N. Mahapatra and S. Dutt, "Random Seeking: A General, Efficient, and Informed Randomized Scheme for Dynamic Load Balancing," *Proc. 10th International Parallel Processing Symposium*, Honolulu, HI, 1996, pp. 881–885.

[18] T. Minyard, Y. Kallinderis, and K. Schulz, "Parallel Load Balancing for Dynamic Execution Environments," *Proc. 34th AIAA Aerospace Sciences Meeting*, Reno, NV, 1996, Paper 96-0295.

[19] L. Oliker, R. Biswas, and R.C. 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.

[20] T.W. Purcell, "CFD and Transonic Helicopter Sound," *Proc. 14th European Rotorcraft Forum*, Milan, Italy, 1988, Paper 2.

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

[22] R.C. Strawn and T.J. Barth, "A Finite-Volume Euler Solver for Computing Rotary-Wing Aerodynamics on Unstructured Meshes," *J. AHS* **38** (1993) 61–67.

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

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

[25] A. Vidwans, Y. Kallinderis, and V. Venkatakrishnan, "A Parallel Dynamic Load Balancing Algorithm for 3-D Adaptive Unstructured Grids," *AIAA J.* **32** (1994) 497–505.

[26] C. Walshaw and M. Berzins, "Dynamic Load-Balancing for PDE Solvers on Adaptive Unstructured Meshes," *Concurrency: Practice and Experience* **7** (1995) 17–28.