PLUM: PARALLEL LOAD BALANCING FOR UNSTRUCTURED ADAPTIVE MESHES

by

LEONID OLIKER

B.S., University of Pennsylvania, 1991

B.S., Wharton School of Business, 1991

M.S., University of Colorado, Boulder, 1994

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

1998

This thesis for the Doctor of Philosophy degree by

Leonid Oliker

has been approved for the

Department of

Computer Science

by

____

Oliver A. Mcbryan

____

Charbel Farhat

Date ______________________

Oliker, Leonid (Ph. D., Computer Science)

PLUM: Parallel Load Balancing for Unstructured Adaptive Meshes

Thesis directed by Professor Oliver A. Mcbryan

Dynamic mesh adaption on unstructured grids is a powerful tool for computing large-scale problems that require grid modifications to efficiently resolve solution features. Unfortunately, an efficient parallel implementation is difficult to achieve, primarily due to the load imbalance created by the dynamically-changing nonuniform grid. To address this problem, we have developed PLUM, an automatic portable framework for performing adaptive large-scale numerical computations in a message-passing environment.

First, we present an efficient parallel implementation of a tetrahedral mesh adaption scheme. Extremely promising parallel performance is achieved for various refinement and coarsening strategies on a realistic-sized domain. Next we describe PLUM, a novel method for dynamically balancing the processor workloads in adaptive grid computations. This research 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 significant runtime improvement is achieved by observing that data movement for a refinement step should be performed after the edge-marking phase but before the actual subdivision. We also present optimal and heuristic remapping cost metrics that can accurately predict the total overhead for data redistribution.

Several experiments are performed to verify the effectiveness of PLUM on sequences of dynamically adapted unstructured grids. Portability is demonstrated by presenting results on the two vastly different architectures of the SP2 and the Origin2000. Additionally, we evaluate the performance of five state-of-the-art partitioning algorithms that can be used within PLUM. It is shown that for certain classes of unsteady adaption, globally repartitioning the computational mesh produces higher quality results than diffusive repartitioning schemes. We also demonstrate that a coarse starting mesh produces high quality load balancing, at a fraction of the cost required for a fine initial mesh. Results indicate that our parallel load balancing strategy will remain viable on large numbers of processors.

ACKNOWLEGEMENTS

First, I would like to express my deepest gratitude to Rupak Biswas without whom this thesis would never have been possible. I was truly fortunate to have such a knowledgeable and generous mentor. Regardless of his busy schedule he always found time to help me on my thesis. I especially would like to thank him for his patience, encouragement, and most importantly his friendship.

I want to thank Oliver McBryan, Charbel Farhat, Xiao-Chuan Cai, and Richard Byrd for serving on my committee.

I am deeply indebted to Roger Strawn. Our collaboration on the prediction and analysis of helicopter noise, provided the starting ground from which my thesis was built.

I was fortunate for the opportunity to work with Robert Schreiber. His wisdom and enthusiasm are an inspiration. One of his many contributions was the development of a theoretical framework for addressing the reassignment problem. I would also like to thank Hal Gabow for taking the time to share his algorithmic insights with me.

I want to thank Andrew Sohn and Horst Simon for their collaboration. Their work greatly contributed to many of the ideas presented in this thesis. I also sincerely thank Vipin Kumar and George Karypis for their help with the MeTiS partitioners, and Chris Walshaw for his help with the Jostle partitioners.

My heartfelt gratitude goes to my parents for their dedication and love. I was incredibly lucky to have their constant support and encouragement. I owe much to Arin Fishkin for her love, proofreading skills, and yummy risotto. Her companionship made my research efforts more enjoyable and productive. Additionally, I thank

Tim Barkow for his editing skills and years of easy living. I also want thank my dear friends Larry Smith, Ted Rheingold, Alex Hart, Sam Boonin, and Eric Hecker for providing me with necessary distractions during my thesis work.

Finally, I would like to thank Joe Oliger for his gracious support and rock climbing beta. This work has been supported by NASA via Contract NAS 2-96027 between NASA and the Universities Space Research Association (USRA). This work was performed at the Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, Moffett Field, CA.

CONTENTS

CHAPTER

1 INTRODUCTION 1

1.1 Thesis Objective 1

1.2 Historical Review 4

1.2.1 Combinatorial Methods 4

1.2.2 Local Diffusive Methods 6

1.2.3 General Global Methods 9

1.2.4 Repartitioning Methods 11

1.3 Thesis Outline 16

2 PARALLEL TETRAHEDRAL MESH ADAPTION 18

2.1 Serial Mesh Adaption Overview 20

2.2 Distributed-Memory Implementation 23

2.2.1 Initialization 24

2.2.2 Execution 26

2.2.3 Finalization 29

2.3 Euler Flow Solver 31

2.4 Experimental Results 32

2.4.1 Refinement Phase 34

2.4.2 Coarsening Phase 38

2.4.3 Initialization and Finalization Phases 39

3 DYNAMIC LOAD BALANCING 41

3.1 Dual Graph of Initial Mesh 42

3.2 Preliminary Evaluation 43

3.3 Parallel Mesh Repartitioning 44
3.4 Similarity Matrix Construction 45
3.5 Processor Reassignment 46
3.5.1 TotalV metric 46
3.5.2 MaxV metric 48
3.5.3 MaxSR metric 50
3.5.4 Heuristic Algorithm 52
3.6 Cost Calculation 54
3.7 Data Remapping 58
3.8 Experimental results 59
4 PORTABILITY AND REPARTITIONING ANALYSIS 73
4.1 Helicopter rotor test case 73
4.1.1 PLUM on the Origin2000 74
4.1.2 The redistribution cost model on the Origin2000 77
4.2 Unsteady simulation test case 79
4.2.1 Comparison of partitioners 82
4.2.2 SP2 vs. Origin2000 86
4.2.3 Coarse vs. fine initial mesh 87
4.2.4 Growing vs. stable mesh 91
5 SUMMARY AND FUTURE WORK 94
5.1 Summary 94
5.2 Future Work 99
BIBLIOGRAPHY 102

TABLES

TABLE

2.1 Progression of Grid Sizes through Refinement and Coarsening for the Different Strategies 33

2.2 Performance of Mesh Refinement when Edges are Bisected Randomly 34

2.3 Performance of Mesh Refinement when Edges are Bisected based on Flow Solution 35

2.4 Performance of "Load-Balanced" Mesh Refinement when Edges are Bisected based on Flow Solution 36

2.5 Quality of Load Balance Before and After Mesh Refinement 37

2.6 Performance of Mesh Coarsening 38

2.7 Performance of Initialization and Finalization Steps for REAL_1R Strategy 39

3.1 Grid sizes for the three different refinement strategies 59

3.2 Comparison of five processor reassignment algorithms for the Real_2R case on the SP2 with $F = 1$. 63

3.3 Progression of Grid Size through a Sequence of Three Levels of Adaption 67

4.1 Execution time of 3D_TAG on the SP2 and the Origin2000 when data is remapped before mesh refinement 76

4.2 Remapping time within PLUM on the SP2 and the Origin2000 when data is redistributed before mesh refinement 79

4.3 Partitioning time on the SP2 for P=64 using a variety of partitioners for Sequence_1 83

4.4 Load imbalance factor before and after mesh partitioning for P=64 using a variety of partitioners for Sequence_1 84

4.5 Percentage of cut edges before and after mesh partitioning for P=64 using a variety of partitioners for Sequence_1 85

4.6 Remapping time on an SP2 for P=64 using the default and our heuristic strategies for Sequence_1 86

4.7 Partitioning and remapping times on the SP2 and the Origin2000 for P=32 using PMeTiS and DAMeTiS for Sequence_1 87

4.8 Load imbalance factor and percentage of cut edges before and after mesh partitioning for P=32 using PMeTiS and DAMeTiS for Sequence_1 88

4.9 Partitioning and remapping times on an SP2 for P=32 using PMeTiS and DAMeTiS for Sequence_3 92

4.10 Load imbalance factor and percentage of cut edges before and after mesh partitioning for P=32 using PMeTiS and DAMeTiS for Sequence_3 93

FIGURES

FIGURE

1.1 Overview of PLUM, our framework for parallel adaptive numerical computation. 2

2.1 Three types of subdivision are permitted for a tetrahedral element. 21

2.2 Sample edge-marking pattern for element subdivision. 22

2.3 An example showing the communication need to form the SPL for a shared vertex. 25

2.4 An example showing how boundary faces are represented at partition boundaries. 26

2.5 A two-dimensional example showing communication during propagation of the edge marking phase. 27

2.6 An example showing how a new edge across a face is classified as shared or internal. 29

2.7 Cut-out view of the initial tetrahedral mesh. 32

3.1 An example of a similarity matrix M for $P = 4$ and $F = 2$. Only the non-zero entries are shown. 45

3.2 Various cost metrics of a similarity matrix M for $P = 4$ and $F = 1$ using (a) optimal MWBG algorithm, (b) optimal BMCM algorithm, (c) optimal DBMCM algorithm, and (d) our heuristic algorithm. 49

3.3 Pseudocode for our heuristic algorithm for solving the processor reassignment problem. 53

3.4 Speedup of the 3D_TAG parallel mesh adaption code when data is remapped either after or before mesh refinement. 60

3.5 Remapping times within PLUM when data is remapped either after or before mesh refinement. 61

3.6 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_2R strategy. 62

3.7 Anatomy of execution times for the REAL_1R, REAL_2R, and REAL_3R refinement strategies. 66

3.8 Maximum (top) and actual (bottom) impact of load balancing on flow solver execution times for different mesh growth factors $\mathcal{G}$. . . 68

3.9 Final adapted mesh and computed pressure contours in the plane of the helicopter rotor. 69

3.10 Anatomy of execution times for the three levels of adaption. 70

3.11 Remapping time as a function of the TotalV (top) and the MaxSR (bottom) metrics. 72

4.1 Speedup of 3D_TAG the Origin2000 when data is remapped either after or before mesh refinement. 75

4.2 Remapping time within PLUM on the the Origin2000 when data is redistributed either after or before mesh refinement. 77

4.3 Anatomy of execution times for the Real_1R, Real_2R, and Real_3R refinement strategies on the Origin2000. 78

4.4 Remapping time as a function of TotalV and MaxSR on the Origin2000. 80

4.5 Initial and adapted meshes (after levels 1 and 5) for the simulated unsteady experiment. 81

4.6 Progression of grid sizes through nine levels of adaption for the unsteady simulation. 82
4.7 PMeTiS partitioning and remapping times using the heuristic strategy for P=16 and 64 on an SP2 for Sequence_1 and Sequence_2. . . 89
4.8 Load imbalance factor and percentage of cut edges after mesh partitioning using PMeTiS for P=16 and 64 for Sequence_1 and Sequence_2. Note that the imbalance factor curves for the two sequences are overlaid. 90

CHAPTER 1

INTRODUCTION

Dynamic mesh adaption on unstructured grids is a powerful tool for computing large-scale problems that require grid modifications to efficiently resolve solution features. By locally refining and coarsening the mesh to capture physical phenomena of interest, such procedures make standard computational methods more cost effective. Unfortunately, an efficient parallel implementation of these adaptive methods is rather difficult to achieve, primarily due to the load imbalance created by the dynamically-changing nonuniform grid. This requires significant communication at runtime, leading to idle processors and adversely affecting the total execution time. Nonetheless, it is generally thought that unstructured adaptive-grid techniques will constitute a significant fraction of future high-performance supercomputing. Various dynamic load balancing methods have been reported to date [17, 18, 20, 21, 22, 37, 42, 67, 70]; however, most of them either lack a global view of loads across processors or do not apply their techniques to realistic large-scale applications.

1.1 Thesis Objective

The purpose of this research effort is to efficiently simulate steady and unsteady aerodynamic flows around realistic engineering-type geometries on multiprocessor systems. The computational cost and memory requirements of large-scale fluid dynamic simulations is prohibitive on classical scalar computers, while vector computers do not seem to keep up with the demands of todays CFD applications [15]. Our thesis objective is to build a portable system for efficiently performing adaptive

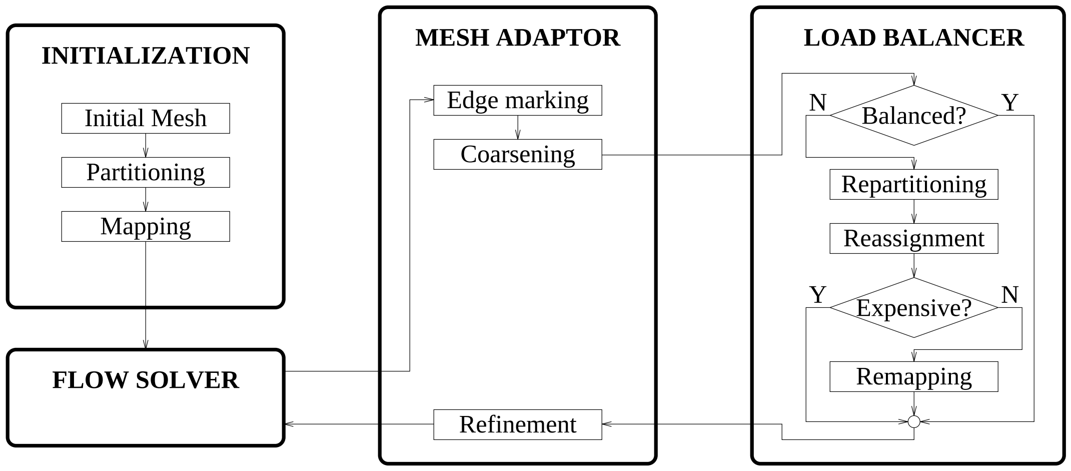


Figure 1.1. Overview of PLUM, our framework for parallel adaptive numerical computation.

large-scale flow calculations in a parallel message-passing environment. Figure 1.1 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 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. 1.1 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.

1.2 Historical Review

The introduction of grid adaption in a parallel environment generally invalidates the initial decomposition, since the computational requirements have changed nonuniformly on each processor. Therefore it is critical that the load be dynamically rebalanced as part of the adaptive calculation procedure. The general problem of dynamic load balancing has been widely studied in the literature and many techniques have been proposed for parallel systems. Their performance depends on several factors in addition to the specific application. These include the interconnection network, the number of processors, and the size of the problem. The abstract goal of load balancing can be stated as follows [73]:

Given a collection of tasks comprising a computation and a set of processors on which these tasks can be executed, find the mapping of tasks to processors that minimize the runtime of the computation.

Various methods of dynamic load balancing have been reported to date, however, most of them lack a global view of loads across processors. Some of these techniques are not scalable, others have only been implemented on toy problems, many theoretical schemes are too complex to reasonably implement, and some methods fail to consider communication locality. A popular approach is to rely on local migration methods where each nodes decisions are based only on local knowledge, and loads are exchanged between neighboring processors. The following section examines some of the dynamic load balancing techniques in the literature.

1.2.1 Combinatorial Methods

One way of performing dynamic load balancing is through general combinatorial techniques such as simulated annealing or genetic algorithms. Simulated Annealing (SA) [46] is a popular heuristic method for finding suboptimal solutions to combinatorial problems. The technique is analogous to a method in statistical mechanics designed to simulate the physical process of annealing. SA simulates the slow cooling of solids as a way to approximate the solutions to combinatorial problems. It works by iteratively proposing new distributions and evaluating their quality. If the new solution is an improvement over the previous iteration that state is accepted. Otherwise the new solution may be chosen according to a probability which decreases as the temperature cools. This process continues until the solution state is frozen and no further improvements can be made. SA requires the user to specify several parameters including the starting temperature and cooling schedule. In general, finding a combination of these parameters to produce a balanced work load in a small amount of time is difficult, because these inputs may differ for each problem.

Genetic Algorithms (GA) [45] are a model of machine learning which derive their behavior from a metaphor based on the processes of natural evolution. It is considered a general and robust optimization method. Briefly, GA starts with an initial population which is typically generated randomly and consists of a set of individuals, or in our case a work load distribution. A set of generic operators are used to generate new individuals from the current population using a process called reproduction, consisting of crossovers and mutations. The basis of GA is that individuals which contribute to the minimization of the object function are more likely to reproduce. Once again, a large number of parameters must be set for a successful distribution.

In general, stochastic optimization techniques on their own are not a popular approach for solving load balancing problems. They can be slow, trapped in local minima, and their behavior depends on many parameters which must be carefully tuned for each application. These methods, however, may be very useful in fine tuning an existing load distribution.

Another combinatorial approach is to use probabilistic techniques. In Random Seeking [49], source processors randomly seek out sink processors for load balancing by flinging probe messages. The probes not only locate sinks, but also collect load distribution information which is used to efficiently regulate load balancing activities. This method works well for certain types of problems such as parallel best-first branch and bound algorithms.

Random Matching [31] is an algorithm based on solving the abstract problem of Incremental Weight Migration on arbitrary graphs, where edge mappings are randomly chosen based only on local information. This is a simple, randomized algorithm which provably results in asymptotically optimal convergence toward a perfect balance. In general these probabilistic techniques are not suitable for balancing adaptive mesh computations. They require too many iterations, could result in disjoint subdomains, ignore edge weights, and send small messages across the network resulting in a high cumulative start up cost overhead.

1.2.2 Local Diffusive Methods

Diffusion is a well know algorithm for load balancing in which tasks model the heat equation by moving from heavily loaded processors to lightly loaded neighbors. A processor's neighbor may be defined by its hardware topology or the connectivity of the distributed domain. Diffusion was first presented as a method for load balancing in [20] and is defined as follows: For a system of P processors, let $w_i(t)$ be the work load on processor i at time t. Adjust the workloads at time $t+1$ as follows:

$$w_i(t+1) = w_i(t) + \sum_{j \in \mathcal{N}(i)} \left(w_j(t) - w_i(t)\right)/2 \qquad (1.1)$$

where $\mathcal{N}(i)$ is the set of all processors connected to processor i. This process can be mapped onto the diffusion equation, and much is known about its properties. In particular, it can be shown that this process will eventually converge. The convergence time, τ, however grows like $\tau \approx \frac{P^2}{3}$ which is rather high.

Kohring [42] presents a simple non-linear variant on the diffusion scheme which considers strip decompositions of the domain. Each processor calculates its own load, by measuring the elapsed CPU-time since the last load balancing step. If a processor finds that one of its neighbors has needed more CPU-time than itself, it transfers one complete row of link-cells to that neighbor. This algorithm shows better convergence properties then the standard diffusion methods.

The basic diffusion algorithm is improved in [73] by using a second-order unconditionally stable differencing scheme. This algorithm improves convergence by allowing larger time steps to be taken without adding substantial complexity. The task transfers are still limited to nearest neighbors in this approach.

Sender Initiated Diffusion (SID) [74] is a highly distributed asynchronous local approach which makes use of nearest neighbor load information to apportion surplus load from heavily loaded processors to underloaded neighbors. Here processors whose loads exceed a certain prespecified threshold, apportion the excess load to deficient neighbors. Receiver Initiated Diffusion (RID) is the converse of the SID strategy in that underloaded processors request loads from overloaded neighbors. For most cases RID has been shown as being a superior approach to SID.

Cyclic Pairwise Exchange is an algorithm presented by Hammond [32] in which processor pairs are defined by the hardware interconnections. Pairwise exchanges of tasks are then performed to iteratively improve an imbalanced load. This method has been shown to improve the mapping time of SA by up to a factor of six. Unfortunately this approach works best for SIMD architectures, and task movements are performed one at a time.

Tiling is another approach to dynamic load balancing originally based on the work of Leiss and Reddy [47]. This procedure is modified by Devine et al. [24]

to migrate finite elements between processors. Each processor is considered a neighborhood center, where a neighbor is defined as that processor and all processors which share its subdomain boundaries. Processors within a given neighborhood are balanced with respect to each other using local performance measurements. Task migration occurs from highly loaded to lightly loaded neighbors within each neighborhood. This iterative process continues until the load is globally balanced. In [23] only one iteration of the tiling algorithm is performed, thereby not achieving a global balance in exchange for speed.

To incorporate more global information, Shephard et al. [58] use a modified Tiling technique where the processors are hierarchically arranged as nodes in a tree. The load is then balanced by iteratively migrating the work from heavily loaded processors through the tree until the load distribution is within a specified tolerance. This methodology has an improved worst case load imbalance over the flat Tiling model if enough iterations are permitted.

We believe that these local iterative techniques are not ideally suited for dynamically balancing unsteady flow calculations. These applications are prone to dramatically shifting the load distribution between adaption phases, causing small regions of the domain to suddenly incur high computational costs. Local diffusion techniques would therefore be required to perform many iterations before global convergence, or accept an unbalanced load in exchange for faster performance. Also, by limiting task movement to nearest neighbors, a finite element may have to make several hops before arriving at its final destination. Current hardware architectures such as the IBM SP2 use wormhole routing making it unnecessary for a unit of work to be moved to more than one processor. Since the remapping must be frequently applied, its cost can become a significant part of the overall performance and must therefore be minimized. By moving large chunks of work units directly to their destinations, the high start up cost of interprocessor communication can be amortized. We therefore assert that there exists a need for balancing strategies which can globally coordinate the distribution of all workloads within the system.

1.2.3 General Global Methods

Many global load balancing approaches are addressed in the literature. The Dimension Exchange Algorithm (DE) is a global technique which steps through each dimension in a hypercube. At each step i a processor exchanges workload with its dimension i neighbor in such a way that their load becomes equal. After $log(P)$ passes, all P processors are guaranteed to have the same workload. DE has been shown to outperform several local schemes [74] including nearest neighbor diffusion and hierarchical balancing methods. This algorithm is ideal for hypercubes and store-and-forward networks, but is not well suited for wormhole routed systems since the global movement of data will usually require multiple hops.

Another approach to global load balancing is based on prefix computations or scans [33].

A <u>scan</u> $(\oplus, V)$ on a vector $V = (V_1, \cdots, V_n)$ with the associative operator $\oplus$ gives as a result the vector of partial results $(I_\oplus, V_1, V_1 \oplus V_2, \cdots, V_1 \oplus V_{n-1})$ where $I_\oplus$ is the identity for $\oplus$.

This operation can be carried out in $O(logP)$ time. Load balancing techniques based on this operation are interesting because they preserve decomposition locality, i.e., given a definition of a neighborhood, tasks which are neighbors before the load balance step will be neighbors afterwards as well.

The algorithm by Baigioni [6] first performs a scan of the load on each processor, from which it calculates the flow. This is defined as the difference between the processor index multiplied by the average work and the value for the scan in that processor. The absolute value of the flow in any particular processor represents the activity that must be moved to another the processor. This algorithm guarantees a perfect load balance, but can only communicate a unit of work one step at a time and is most suitable for SIMD architectures. A variation of this algorithm called Position Scan Load Balancing (PSLB), communicates the work directly to the destination processor, making it more suitable for MIMD systems [33]. This methodology is currently limited to structured grids and does not consider subdomain boundary quality.

A theoretical global technique by Bogleav [14] uses linear programming algorithms to exactly load balance tasks on arbitrary topologies. This solution is computed using the simplex method which is considered a fast and accurate optimization technique. Unfortunately the computation time is polynomial in the number of elements which makes it prohibitively expensive within our framework.

Index-based algorithms are another approach to the partitioning problem presented by Ou, Ranka, and Fox [51]. First, vertices of a graph are mapped onto one dimensional list, which is then distributed among the processors by assigning contiguous blocks of vertices to each partition. When the computational load changes, the graph can be remapped by repartitioning the one-dimensional list. This requires calculating the indices of the new vertices and combining them with the vertices of the original list, which corresponds to merging an unsorted list of integers to a sorted list. This operation can then be performed quickly in parallel. Unfortunately, the index-based algorithms assume that only small perturbations are made in the load, which does not hold true for unsteady flow problems. Subdomain interface quality is also inferior to other methods, since mapping a three dimensional grid onto a one dimension list results in degradation of boundary information.

In [58] an integrated system is built in a parallel framework which includes: mesh generation, equation solution, mesh enrichment, mesh migration, and load balancing. To date, this work mostly closely resembles our efforts. Here, two load balancing schemes are compared in an adaptive grid calculation on a 128 node IBM SP2. The first is a global repartitioning scheme based on a parallel version of Inertial Recursive Bisection (PIRB) while the second is the more iterative approach of hierarchical tiling. PIRB has two advantages over IRB [44] in a parallel setting: its execution time decreases as the number of processors decrease; and the distributed mesh no longer needs to be gathered on one processor before the partitioning phase begins, which can become an expensive operation in both time and space as the mesh grows. The preliminary test results indicate that the iterative load migration scheme tends to be more computationally expensive than the global PIRB algorithm, while at the same time yielding lower quality subdomains. Although these tests are by no means exhaustive, they do support our claim that a global methodology is the superior approach for addressing dynamic load balancing on these types of problems.

1.2.4 Repartitioning Methods

It usually considered too expensive to repartition the entire domain in the inner loop of adaptive flow calculations, due to the potentially high partitioning and data movement cost. Some dynamic load balancing techniques reuse the original partition by only considering the transfer of those elements located on the subdomains boundaries. In the work of Vanderstraeten et al. [69] a decomposed domain undergoes one level of adaptive refinement resulting in an unbalanced load. A comparison is then made between retrofitting the original decomposition along its boundaries (using SA) and performing the decomposition from scratch (using the Greedy technique of Farhat [57] followed by SA). The results indicate that the latter technique performed faster, contained higher quality subdomains, and required fewer element exchanges between partitions. Since the adaption phase created many new elements in a small region, as is common in unsteady flows, the original decomposition is not necessarily a good starting point for the retrofitting approach. Retrofitting is only useful when a small percentage of the elements are refined in a consistent manner throughout the previously generated subdomains.

Many heuristics have been developed for graph partitioning since the optimal solution is an NP-hard problem [30]. Spectral bisection algorithms [25, 26] are a class of partitioning techniques developed in the early 1970's which are known to produce high quality subdomains for a wide class of problems. These ideas were extended in Recursive Spectral Bisection (RSB) by Simon [61] for partitioning finite element meshes. Unfortunately, spectral methods are considered too expensive to be performed within the inner loop of time critical computations. This is especially true when the domain size grows in an adaptive refinement, since computing the Fiedler vector for a problem of size n, is $O(n\sqrt{n})$ [2]. Several attempts have been made to integrate spectral techniques with dynamic load balancing. Walshaw and Berzins [74] propose a method called Dynamic Recursive Spectral Bisection (DRSB), which limits the repartitioning time by clustering internal vertices and only allowing boundary elements to move across partitions. In other words, mesh elements which are far enough away from an interprocessor boundary will be ignored during the repartitioning phase, resulting in a clusters of mesh elements separated by a strip of elements along the boundaries. The spectral partitioning algorithm then proceeds on the reduced size graph, under the assumption that clustered nodes will remain in their original partitions. This technique is only applicable under the assumption that there will be a small change in the domain size, otherwise it reverts back to the standard RSB method.

In [68] Driessche and Roose propose extending the (recursive) spectral bisection algorithm so that it applies to dynamically changing grids. They propose a repartitioning technique which not only ensures that the grid subdomains are equally sized with short interfaces, but attempts to minimize the cost of element transfers across partition boundaries. Traditional spectral techniques do not incorporate this last component, which can be a very costly operation. This more complex problem is modeled as a partitioning problem, by extending the original grid with virtual edges and virtual vertices. One virtual vertex is added to each partition with virtual edges added between the virtual vertex and the vertices that correspond to the grid points that were originally assigned to that processor. The weight of a virtual edge is equal to the cost of transferring the corresponding grid point to another processor. A partition of the extended graph not only cuts ordinary edges but also a number of virtual ones, thereby modeling both the application communication cost and the element transfer cost. The run time of this method is comparable to traditional spectral algorithms, but due to the extension, several iterations of the new partitioner must be executed to achieve a perfect load balance.

The HARP [60] repartitioner has recently been proposed as a method for balancing adaptive grids. This new algorithm is based on the observation that for most discretized bodies, a significant portion of their structure can usually be captured with only a few of their eigenvectors. Therefore, a preprocessing step computes and stores the appropriate number of eigenpairs. In order for these values to remain valid, the connectivity of the graph must remain the same throughout the computation. This can be achieved by adding weights to the vertices of the original graph, as elements become refined. Once the flow computation starts, the Fiedler vector no longer needs to be computed at each iteration, resulting in partitioning times which are several orders of magnitude faster than RSB. Note that since the connectivity of the graph remains the same, the partitioner must assume that edge weights do not change throughout the course of the computation. The impact of this restriction is application specific.

Multilevel algorithms [34, 36, 40, 71] present a way to reduce the computational requirement of partitioning, while maintaining high quality subdomains. These algorithms reduce the size of the graph by collapsing vertices and edges. The smaller graph is then partitioned, and the results are uncoarsened to construct a partition for the original graph. The most sophisticated schemes use several stages of contraction and uncoarsening, and smooth the graph during the latter phase. It has been shown [36] that for a variety of finite element problems, multilevel schemes can provide higher partitioning quality than spectral methods at a lower cost. Chaco [34], MeTiS [40], and Jostle [71] are three popular software package which provide several powerful partitioning options.

Recently, several parallel multilevel schemes have become available. An advantage of these algorithms is that they are fast enough to be included in the inner loop of adaptive flow calculations. PMeTiS [41] and Jostle-MS [72] are parallel, multilevel, k-way partitioning codes. They are considered global algorithms since they make no assumptions on how the graph is initially distributed among the processors. PMeTiS uses a greedy graph growing algorithm for partitioning the coarsest graph, and uncoarsens it by using a combination of boundary greedy and Kernighan-Lin [43] refinement. Jostle-MS uses a greedy algorithm to partition the coarsest graph followed by a parallel iterative scheme based on relative gain to optimize each of the multilevel graphs.

UAMeTiS [62], DAMeTiS [62], and Jostle-MD [72] are diffusive multilevel schemes which are designed to repartition adaptively refined meshes by modifying the existing partitions. Reported results indicate that these algorithms produce partitions of quality comparable to that of their global counterparts, while dramatically reducing the amount of data that needs to be moved due to repartitioning. UAMeTiS and DAMeTiS perform local multilevel coarsening followed by multilevel diffusion and refinement to balance the graphs while maintaining the edge-cut. The difference between these two algorithms is that UAMeTiS performs undirected diffusion based on local balancing criteria, whereas DAMeTiS uses a 2-norm minimization algorithm at the coarsest graph to guide the diffusion, and is thus considered directed. Jostle-MD performs graph reduction on the existing partitions, followed by the optimization techniques used in Jostle-MS. One major difference between these diffusive algorithms is that Jostle-MD employs a single level diffusion scheme, while UAMeTiS and DAMeTiS use multilevel diffusion. An extensive performance analysis of the MeTiS and Jostle partitioners within PLUM is presented in Chapters 3 and 4.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2, we present our parallel implementation of a tetrahedral mesh adaption code. The parallel version consists of C++ and MPI code wrapped around the original serial mesh adaption program of Biswas and Strawn [12]. An object-oriented approach allowed a clean and efficient implementation. Experiments are performed on a realistic-sized computational mesh used for a helicopter acoustics simulation. Results show extremely promising parallel performance on 64 processors of an IBM-SP2.

Chapter 3 presents PLUM, an automatic portable framework for performing adaptive numerical computations in a message passing environment. We describe the implementation and integration of all major components within our dynamic load balancing system. Several salient features of PLUM are 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. Results demonstrate that PLUM is an effective dynamic load balancing strategy which remains viable on a large number of processors.

Chapter 4 presents several experimental results that verify the effectiveness of PLUM on sequences of dynamically adapted unstructured grids. We examine portability by comparing results between the distributed-memory system of the IBM SP2 and the Scalable Shared-memory MultiProcessing (S2MP) architecture of the SGI/Cray Origin2000. Additionally, we evaluate the performance of five state-of-the-art partitioning algorithms that can be used within PLUM. Results indicate that a global repartitioner can outperform diffusive schemes in both subdomain quality and remapping overhead. Finally, we demonstrate that PLUM works well for both for both steady and unsteady adaptive problems with many levels of adaption, even when using a coarse initial mesh. A finer starting mesh may be used to achieve lower edge cuts and marginally better load balanceing. but is generally not worth the increased partitioning and data remapping times.

Chapter 5 contains a summary of our work, and some future directions for this research.

CHAPTER 2

PARALLEL TETRAHEDRAL MESH ADAPTION

Accurate simulation of the evolution of steady and unsteady aerodynamic flows around complex bodies is a common challenge in many fields of computational fluid dynamics. The unstructured discretization of the flow domain is an effective way for dealing with the complex geometries and moving bodies. Hyperbolic PDEs are dominated by the propagation and interaction of waves, which occupy a small portion of the problem domain. Therefore the advantage of solutions on unstructured grids in comparison to structured ones, is the excellent flexibility of adapting the mesh to the local requirements of the solution. The drawbacks are the relatively high demands on computational time and storage. This can be compensated by using fine grids to represent the relatively small regions occupied by flow field phenomena, while representing the remaining regions with coarser grids. These savings in storage and CPU requirements typically range between 50-100 compared to an overall fine mesh [48] for a given spatial accuracy.

Two solution-adaptive strategies are commonly used with unstructured-grid methods. Regeneration schemes generate a new grid with a higher or lower concentration of points in different regions depending on an error indicator. A major disadvantage of such schemes is that they are computationally expensive. This is a serious drawback for unsteady problems where the mesh must be frequently adapted. However, resulting grids are usually well-formed with smooth transitions between regions of coarse and fine mesh spacing.

Local mesh adaption, on the other hand, involves adding points to the existing grid in regions where the error indicator is high, and removing points from regions where the indicator is low. The advantage of such strategies is that relatively few mesh points need to be added or deleted at each refinement/coarsening step for unsteady problems. However, complicated logic and data structures are required to keep track of the points that are added and removed.

For problems that evolve with time, local mesh adaption procedures have proved to be robust, reliable, and efficient. By redistributing the available mesh points to capture flowfield phenomena of interest, such procedures make standard computational methods more cost effective. Highly localized regions of mesh refinement are required in order to accurately capture shock waves, contact discontinuities, vortices, and shear layers. This provides scientists the opportunity to obtain solutions on adapted meshes that are comparable to those obtained on globally-refined grids but at a much lower cost.

Advances in adaptive software and methodology notwithstanding, parallel computational strategies will be an essential ingredient in solving complex real-life problems. However, parallel computers are easily programmed with regular data structures; so the development of efficient parallel adaptive algorithms for unstructured grids poses a serious challenge. Their parallel performance for supercomputing applications not only depends on the design strategies, but also on the choice of efficient data structures which must be amenable to simple manipulation without significant memory contention (for shared-memory architectures) or communication overhead (for message-passing architectures).

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 adaptive schemes have been successfully developed in a parallel environment. Most of these codes are based on two-dimensional finite elements [3, 4, 7, 9, 16, 38, 39, 55], and some progress has been made towards three-dimensional unstructured-mesh simulations [8, 50, 56, 58].

This chapter presents an efficient parallel implementation of a dynamic mesh adaption code [12] which has shown good sequential performance. The parallel version consists of an additional 3,000 lines of C++ with Message-Passing Interface (MPI), allowing portability to any system supporting these languages. This code is a wrapper around the original mesh adaption program written in C, and requires almost no changes to the serial code. Only a few lines were added to link it with the parallel constructs. An object-oriented approach allowed this to be performed in a clean and efficient manner.

2.1 Serial Mesh Adaption Overview

We give a brief description of the tetrahedral mesh adaption scheme [12] that is used in this work to better explain the modifications that were made for the distributed-memory implementation. The code, called 3D_TAG, has its data structures based on edges that connect the vertices of a tetrahedral mesh. This means that the elements and boundary faces are defined by their edges rather than by their vertices. These edge-based data structures make the mesh adaption procedure capable of efficiently performing anisotropic refinement and coarsening. A successful data structure must contain the right amount of information to rapidly reconstruct the mesh connectivity when vertices are added or deleted while having reasonable memory requirements.

Recently, the 3D_TAG code has been modified to refine and coarsen hexahedral meshes [13]. The data structures and serial implementation for the hexahedral scheme are similar to those for the tetrahedral code. Their parallel implementations should also be similar; however, this chapter focuses solely on tetrahedral mesh adaption.

At each mesh adaption step, individual edges are marked for coarsening, refinement, or no change, based on an error indicator calculated from the flow solution. Edges whose error values exceed a user-specified upper threshold are targeted for subdivision. Similarly, edges whose error values lie below another user-specified lower threshold are targeted for removal. Only three subdivision types are allowed for each tetrahedral element and these are shown in Fig. 2.1. The 1:8 isotropic subdivision is implemented by adding a new vertex at the mid-point of each of the six edges. The 1:4 and 1:2 subdivisions can result either because the edges of a parent tetrahedron are targeted anisotropically or because they are required to form a valid connectivity for the new mesh. When an edge is bisected, the solution quantities are linearly interpolated at the mid-point from its two end-points.

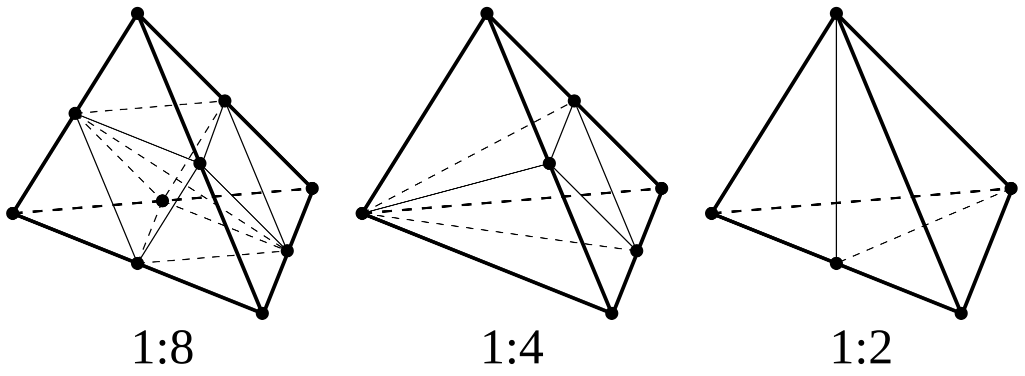


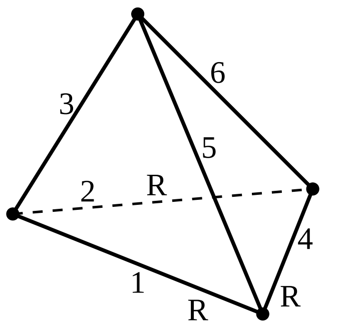
Figure 2.1: Three types of subdivision are permitted for a tetrahedral element.

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 as shown in Fig. 2.2 where the edges marked with an R are the ones to be bisected. 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 [12].

Details of the data structures are given in [12]; however, a brief description of the salient features is necessary to understand the distributed-memory implementation of the mesh adaption code. Pertinent information is maintained for the vertices, elements, edges, and boundary faces of the mesh. For each vertex, the coordinates are stored in `coord[3]`, the flow solution in `soln[5]`, and a pointer to the first entry in the edge sublist in `edges`. The edge sublist for a vertex contains pointers to all the edges that are incident upon it. Such sublists eliminate extensive



6	5	4	3	2	1	Edge number
0	0	1	0	1	1	Pattern = 11

Figure 2.2: Sample edge-marking pattern for element subdivision.

searches and are crucial to the efficiency of the overall adaption scheme. The tetrahedral elements have their six edges stored in `tedge[6]`, the edge-marking pattern in `patt`, the parent element in `tparent`, and the first child element in `tchild`. Sibling elements always reside contiguously in memory; hence, a parent element only needs a pointer to the first child. For each edge, we store its two end-points in `vertex[2]`, its parent edge in `eparent`, its two children edges in `echild[2]`, the two boundary faces it defines in `bfac[2]`, and a pointer to the first entry in the element sublist in `elems`. The element sublist for an edge contains pointers to all the elements that share it. Finally, for each boundary face, we store the three edges in `bedge[3]`, the element to which it belongs in `belem`, the parent in `bparent`, and the first child in `bchild`. Sibling boundary faces, like elements, are stored consecutively in memory.

2.2 Distributed-Memory Implementation

The parallel implementation of the 3D_TAG mesh adaption code consists of three phases: initialization, execution, and finalization. The initialization step consists of scattering the global data across the processors, defining a local numbering scheme for each object, and creating the mapping for objects that are shared by multiple processors. The execution step runs a copy of 3D_TAG on each processor that refines or coarsens its local region, while maintaining a globally-consistent grid along partition boundaries. Parallel performance is extremely critical during this phase since it will be executed several times during a flow computation. Finally, a gather operation is performed in the finalization step to combine the local grids into one global mesh. Locally-numbered objects and the corresponding pointers are reordered to represent one single consistent mesh.

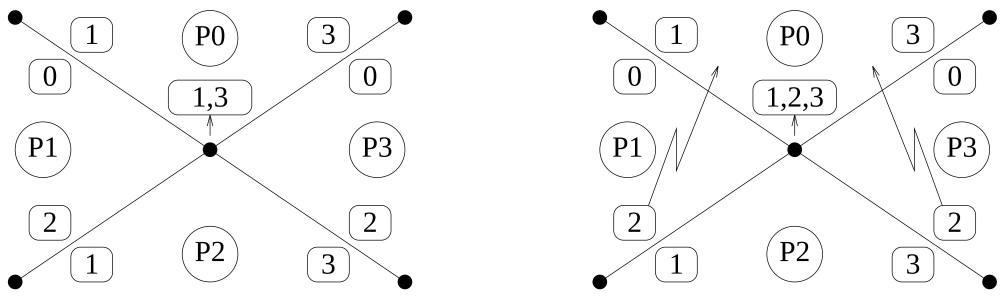
In order to perform parallel mesh adaption, the initial grid must first be partitioned among the available processors. A good partitioner should divide the grid into equal pieces for optimal load balancing, while minimizing the number of edges along partition boundaries for low interprocessor communication. It is also important that within our framework, the partitioning phase be performed rapidly. Some excellent parallel partitioning algorithms are now available [40, 58, 60, 70]; however, we need one that is extremely fast while giving good load balance and low edge cuts. For this set of experiments the parallel MeTis (PMeTiS) partitioner of Karypis and Kumar [40] was used. The PMeTiS algorithm is briefly described in Sec. 1.2.4, and a detailed analysis of its performance is presented in Secs. 3.8 and 4.2.

2.2.1 Initialization

The initialization phase takes as input the global initial grid and the corresponding partitioning that maps each tetrahedral element to exactly one partition. The element data and partition information are then broadcast to all processors which, in parallel, assign a local, zero-based number to each element. Once the elements have been processed, local edge information can be computed.

In three dimensions, an individual edge may belong to an arbitrary number of elements. Since each element is assigned to only one partition, it is theoretically possible for an edge to be shared by all the processors. For each partition, a local zero-based number is assigned to every edge that belongs to at least one element. Each processor then redefines its elements in `tedge[6]` in terms of these local edge numbers. Edges that are shared by more than one processor are identified by searching for elements that lie on partition boundaries. A bit flag is set to distinguish between shared and internal edges. A list of shared processors (SPL) is also generated for each shared edge. Finally, the element sublist in `elems` for each edge is updated to contain only the local elements.

The vertices are initialized using the `vertex[2]` data structure for each edge. Every local vertex is assigned a zero-based number in each partition. Next the local edge sublist for each vertex is created from the appropriate subset of the global edges array. Like shared edges, each shared vertex must be identified and assigned its SPL. A naive approach would be to thread through the data structures to the elements and their partitions to determine which vertices lie on partition boundaries. A faster approach is based on the following two properties of a shared vertex: it must be an end-point for at least one shared edge, and its SPL is the union of its shared edges' SPLs. However, some communication is required when using this method. An example is shown in Fig. 2.3 where the SPL is being formed in P0 for the center vertex that is shared by three other processors. Without communication, P0 would incorrectly conclude that the vertex is shared only with P1 and P3. For each vertex containing a shared edge in its edges sublist, that edge's SPL is communicated to the processors in the SPLs of all other shared edges until the union of all the SPLs is formed. For the cases in this paper, this process required no more than three iterations, and all shared vertices were processed as a function of the number of shared edges plus a small communication overhead.



Before communication
P0 shares center vertex with P1, P3

After communication
P0 shares center vertex with P1, P2, P3

Figure 2.3. An example showing the communication need to form the SPL for a shared vertex.

The final step in the initialization phase is the local renumbering of the external boundary faces. Since a boundary face belongs to only one element, it is never shared among processors. Each boundary face is defined by its three edges in bedge[3], while each edge maintains a pair of pointers in bfac[2] to the boundary faces it defines. Since the global mesh is closed, an edge on the external boundary is shared by exactly two boundary faces. However, when the mesh is partitioned, this is no longer true. An example is shown in Fig. 2.4. An affected edge creates an empty ghost boundary face in each of the two processors for the execution phase which is later eliminated during the finalization stage.

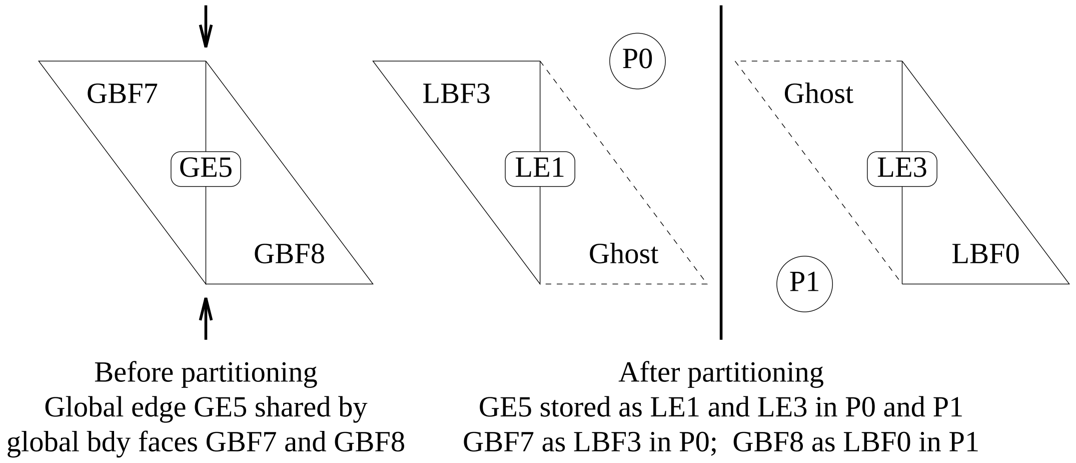

GBF7
GE5
GBF8
LBF3
LE1
Ghost
P0
Ghost
LE3
LBF0
P1
Before partitioning
Global edge GE5 shared by
global bdy faces GBF7 and GBF8
After partitioning
GE5 stored as LE1 and LE3 in P0 and P1
GBF7 as LBF3 in P0; GBF8 as LBF0 in P1

Figure 2.4. An example showing how boundary faces are represented at partition boundaries.

A new data structure has been added to the serial code to represent all this shared information. Each shared edge and vertex contains a two-way mapping between its local and its global numbers, and a SPL of processors where its shared copies reside. The maximum additional storage depends on the number of processors used and the fraction of shared objects. For the cases in this chapter, this was less than 10% of the memory requirements of the serial version.

2.2.2 Execution

The first step in the actual mesh adaption phase is to target edges for refinement or coarsening. This is usually based on an error indicator for each edge that is computed from the flow solution. This strategy results in a symmetrical marking of all shared edges across partitions since shared edges have the same flow and geometry information regardless of their processor number. However, elements have to be continuously upgraded to one of the three allowed subdivision patterns shown in Fig. 2.1. This causes some propagation of edges being targeted that could mark local copies of shared edges inconsistently. This is because the local geometry and marking patterns affect the nature of the propagation. Communication is therefore required after each iteration of the propagation process. Every processor sends a list of all the newly-marked local copies of shared edges to all the other processors in their SPLs. This process may continue for several iterations, and edge markings could propagate back and forth across partitions.

Figure 2.5 shows a two-dimensional example of two iterations of the propagation process across a partition boundary. The process is similar in three dimensions. Processor P0 marks its local copy of shared edge GE1 and communicates that to P1. P1 then marks its own copy of GE1, which causes some internal propagation because element marking patterns must be upgraded to those that are valid. Note that P1 marks its third internal edge and its local copy of shared edge GE2 during this phase. Information about the shared edge is then communicated to P0, and the propagation phase terminates. The four original triangles can now be correctly subdivided into a total of 12 smaller triangles.

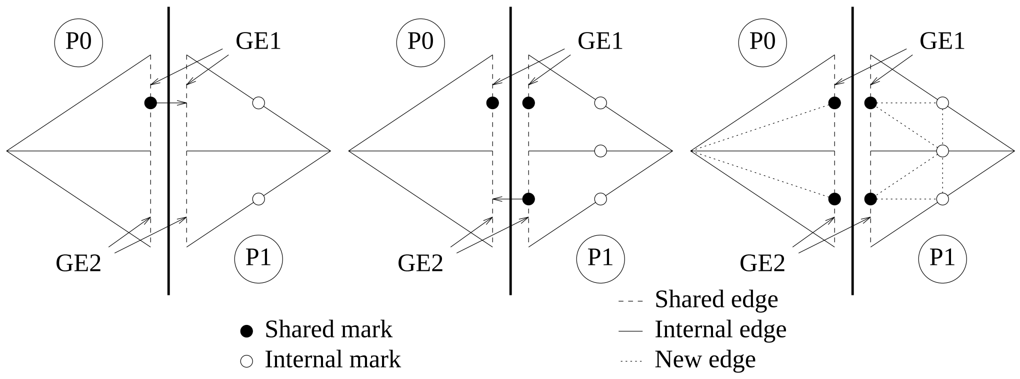


Figure 2.5. A two-dimensional example showing communication during propagation of the edge marking phase.

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 during refinement are assigned shared processor information that depends on several factors. Four different cases can occur when new edges are created.

- If an internal edge is bisected, the center vertex and all new edges incident on that vertex are also internal to the partition. Shared processor information is not required in this case.
- If a shared edge is bisected, its two children and the center vertex inherit its SPL, since they lie on the same partition boundary.
- If a new edge is created in the interior of an element, it is internal to the partition since processor boundaries only lie along element faces. Shared processor information is not required.
- If a new edge is created that lies across an element face, communication is required to determine whether it is shared or internal. If it is shared, the SPL must be formed.

All the cases are straightforward, except for the last one. If the intersection of the SPLs of the two end-points of the new edge is null, the edge is internal. Otherwise, communication is required with the shared processors to determine whether they have a local copy of the edge. This communication is necessary because no information is stored about the faces of the tetrahedral elements. An alternate solution would be to incorporate faces as an additional object into the data structures, and maintaining it through the adaption. However, this does not compare favorably in terms of memory or CPU time to a single communication at the end of the refinement procedure.

Figure 2.6 shows the top view of a tetrahedron in processor P0 that shares two faces with P1. In P0, the intersection of the SPLs for the two end-points of all the three new edges LE1, LE2, and LE3 yields P1. However, when P0 communicates this information to P1, P1 will only have local copies corresponding to LE1 and LE2. Thus, P0 will classify LE1 and LE2 as shared edges but LE3 as an internal edge.

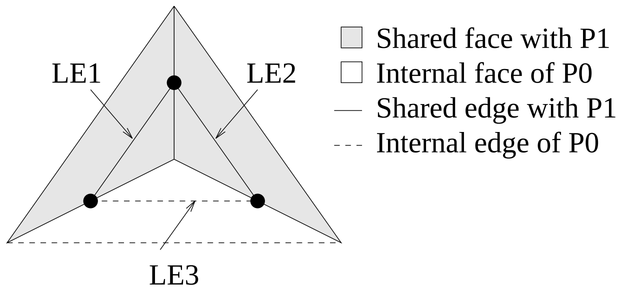


Figure 2.6. An example showing how a new edge across a face is classified as shared or internal.

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

2.2.3 Finalization

Under certain conditions, it is necessary to create a single global mesh after one or more adaption steps. Some post processing tasks, such as visualization, need to processes the whole grid simultaneously. Storing a snapshot of a grid for future restarts could also require a global view. Our finalization phase accomplishes this goal by connecting the individual subgrids into one global data structure.

Each local object is first assigned a unique global number. Next, all local data structures are updated in terms of these global numbers. Finally, gather operations are performed to a host processor to create the global mesh. Individual processors are responsible for correctly arranging the data so that the host only collects and concatenates without further processing.

It is relatively simple to assign global element numbers since elements are not shared among processors. By performing a scan-reduce add on the total number of elements, each processor can assign the final global element number. The global boundary face numbering is also done similarly since they too are not shared among processors.

Assigning global numbers to edges and vertices is somewhat more complicated since they may be shared by several processors. Each shared edge (and vertex) is assigned an owner from its SPL which is then responsible for generating the global number. Owners are randomly selected to keep the computation and communication loads balanced. Once all processors complete numbering their edges (and vertices), a communication phase propagates the global values from owners to other processors that have local copies.

After global numbers have been assigned to every object, all data structures are updated to contain consistent global information. Since elements and boundary faces are unique in each processor, no duplicates exist. All unowned edge copies are removed from the data structures, which are then compacted. However, the element sublists in `elems` cannot be discarded for the unowned edges. Some communication is required to adjust the pointers in the local sublists so that global sublists can be formed without any serial computation. The pair of pointers in `bfac[2]` that were split during the initialization phase for shared edges are glued back by communicating the boundary face information to the owner. Vertex data structures are updated much like edges except for the manner in which their edge sublists in `edges` are handled. Since shared vertices may contain local copies of the same global edge in their sublists on different processors, the unowned edge copies are first deleted. Pointers are next adjusted as in the `elems` case with some communication among processors.

At this time, all processors have updated their local data with respect to their relative positions in the final global data structures. A gather operation by a host processor is performed to concatenate the local data structures. The host can then interface the global mesh directly to the appropriate post-processing module without having to perform any serial computation.

2.3 Euler Flow Solver

An important component of the mesh adaption procedure 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 [64] 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 3D_TAG 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 [13]. This is also true for our dynamic load balancing procedure.

2.4 Experimental Results

The parallel 3D_TAG procedure was originally implemented on the wide node IBM SP2 distributed-memory multiprocessor located at NASA Ames Research Center. The code is written in C and C++, with the parallel activities in MPI for portability. Note that no SP2-specific optimizations were used to obtain the performance results reported in this section. Portability results are presented in Chapter 4.

The computational mesh is the one used to simulate the acoustics experiment of Purcell [54] 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 [65]. This chapter reports only on the performance of the distributed-memory version of the mesh adaption code. A cut-out view of the initial tetrahedral mesh is shown in Fig 2.7.

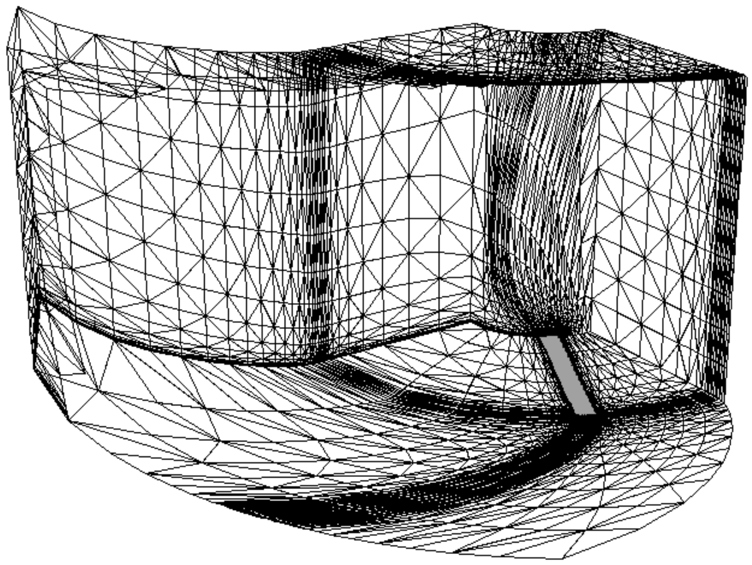


Figure 2.7: Cut-out view of the initial tetrahedral mesh.

Performance results for the parallel code are presented for one refinement and one coarsening step using various edge-marking strategies. Six strategies are used for the refinement step. The first set of experiments, denoted as RANDOM_1R, RANDOM_2R, and RANDOM_3R, consists of randomly bisecting 5%, 33%, and 60% of the edges in the mesh, respectively. The second set, denoted as REAL_1R, REAL_2R, and REAL_3R, consists of bisecting the same numbers of edges using an error indicator [65] derived from the actual flow solution described in Sec. 2.3. These strategies represent significantly different scenarios. In general, the RANDOM cases are expected to behave somewhat ideally because the computational loads are automatically balanced.

Table 2.1. Progression of Grid Sizes through Refinement and Coarsening for the Different Strategies

	Vertices	Elements	Edges	Bdy Faces
Initial Mesh	13,967	60,968	78,343	6,818
REFINEMENT				
RANDOM_1R	18,274	82,417	104,526	7,672
REAL_1R	17,880	82,489	104,209	7,682
RANDOM_2R	39,829	201,734	246,949	10,774
REAL_2R	39,332	201,780	247,115	12,008
RANDOM_3R	60,916	320,919	389,686	15,704
REAL_3R	61,161	321,841	391,233	16,464
COARSENING				
RANDOM_2C	21,756	100,537	126,448	8,312
REAL_2C	20,998	100,124	125,261	8,280

Since the coarsening procedure and performance are similar to the refinement method, only two cases are presented where 7% of the edges in the refined meshes obtained with the RANDOM_2R and the REAL_2R strategies are respectively coarsened randomly (RANDOM_2C) or based on actual flow solution (REAL_2C). Table 2.1 presents the progression of grid sizes through the two adaption steps for each edge-marking strategy.

2.4.1 Refinement Phase

Table 2.2 presents the timings and parallel speedup for the refinement step with the random marking of edges (strategies RANDOM_1R, RANDOM_2R, and RANDOM_3R). Performance is excellent with efficiencies of more than 83% on 32 processors and 76% on 64 processors for the RANDOM_3R case. Parallel mesh refinement shows a markedly better performance for RANDOM_3R due to its bigger computation-to-communication ratio. In general, the total speedup will improve as the size of the refined mesh increases. This is because the mesh adaption time will increase while the percentage of elements along processor boundaries will decrease.

Table 2.2: Performance of Mesh Refinement when Edges are Bisected Randomly

		RANDOM_1R			RANDOM_2R			RANDOM_3R		
P	Edges Shared	Cmp Time	Cmm Time	Spd Up	Cmp Time	Cmm Time	Spd Up	Cmp Time	Cmm Time	Spd Up
1	0.0%	7.044	0.000	1.00	26.904	0.000	1.00	45.015	0.000	1.00
2	1.9%	3.837	0.001	1.84	13.878	0.002	1.94	22.762	0.003	1.98
4	3.7%	2.025	0.002	3.48	7.605	0.004	3.54	11.569	0.004	3.89
8	6.6%	1.068	0.003	6.58	4.042	0.006	6.65	5.913	0.006	7.61
16	8.8%	0.587	0.007	11.86	2.293	0.013	11.67	3.191	0.008	14.07
32	11.6%	0.330	0.010	20.72	1.338	0.022	19.78	1.678	0.013	26.62
64	15.3%	0.191	0.023	32.92	0.711	0.040	35.82	0.896	0.029	48.66

Notice also from Table 2.2 that the communication time is less than 3% of the total time for up to 32 processors for all three cases. On 64 processors, the communication time although still quite small, is only an order of magnitude smaller than the computation time for RANDOM_1R. This begins to adversely affect the parallel speedup and indicates that the saturation point has been reached for this case in terms of the number of processors that should be used. Each partition contains less than 1,000 elements with more than 15% of the edges on partition boundaries when 64 processors are used. Since additional work and storage are necessary for shared edges, the speedup deteriorates as the percentage of such edges increases. The situation is much better for RANDOM_3R since the computation time is significantly higher.

Table 2.3. Performance of Mesh Refinement when Edges are Bisected based on Flow Solution

		REAL_1R			REAL_2R			REAL_3R		
P	Edges Shared	Cmp Time	Cmm Time	Spd Up	Cmp Time	Cmm Time	Spd Up	Cmp Time	Cmm Time	Spd Up
1	0.0%	5.902	0.000	1.00	23.780	0.000	1.00	41.702	0.000	1.00
2	1.9%	3.979	0.002	1.48	18.117	0.003	1.31	26.317	0.003	1.58
4	3.7%	2.530	0.002	2.33	9.173	0.002	2.59	14.266	0.002	2.92
8	6.6%	1.589	0.003	3.71	7.091	0.004	3.35	8.430	0.003	4.95
16	8.8%	1.311	0.006	4.48	4.046	0.006	5.87	4.363	0.004	9.55
32	11.6%	0.879	0.009	6.65	2.277	0.010	10.40	2.278	0.007	18.25
64	15.3%	0.616	0.024	9.22	1.224	0.017	19.16	1.148	0.012	35.95

Table 2.3 shows the timings and speedup when edges are marked using an actual flow solution-based error indicator. Performance is extremely poor, especially for REAL_1R and REAL_2R, with speedups of only 9.2X and 19.2X on 64 processors, respectively. This is because mesh adaption for practical problems occurs in a localized region, causing an almost worst case load-balance behavior. Elements are targeted for refinement on only a small subset of the available processors. Most of the processors remain idle since none of their assigned elements need to be refined. Performance is somewhat better for the REAL_3R strategy since the refinement region is much larger. Since 60% of all edges are bisected in this case, most of the processors are busy doing useful work. This is reflected by an efficiency of more than 56% on 64 processors.

Note also from Table 2.3 that the communication times constitute a much smaller fraction of the total time compared to the results in Table 2.2. This is due to the difference in the distribution of bisected edges. The RANDOM cases require significantly more communication among processors at the partition boundaries because refinement is scattered all over the problem domain. The REAL cases, on the other hand, require much less communication since the refined regions are localized and mostly contained within partitions.

Poor parallel performance of the mesh refinement code for the three Real strategies is due to severe load imbalance. It is therefore worthwhile trying to load balance this phase of the mesh adaption procedure as much as possible. This can be achieved by splitting the mesh refinement step into two distinct phases of edge marking and mesh subdivision. After edges are marked for bisection, it is possible to exactly predict the new refined mesh before actually performing the subdivision phase. The mesh is repartitioned if the edge markings are skewed beyond a specified tolerance. All necessary data is then appropriately redistributed and the mesh elements are refined in their destination processors. This enables the subdivision phase to perform in a more load-balanced fashion. Additionally, a smaller volume of data has to be moved around since remapping is performed before the mesh grows in size due to refinement. A performance analysis of the remapping procedure is presented in Chapters 3 and 4.

Table 2.4. Performance of "Load-Balanced" Mesh Refinement when Edges are Bisected based on Flow Solution

	Real_1R			Real_2R			Real_3R		
P	Cmp Time	Cmm Time	Spd Up	Cmp Time	Cmm Time	Spd Up	Cmp Time	Cmm Time	Spd Up
1	5.902	0.000	1.00	23.780	0.000	1.00	41.702	0.000	1.00
2	3.311	0.001	1.78	12.059	0.001	1.97	21.592	0.001	1.93
4	1.980	0.001	2.98	6.733	0.001	3.53	10.975	0.002	3.80
8	1.369	0.003	4.30	3.430	0.004	6.92	5.678	0.004	7.34
16	0.702	0.006	8.34	1.840	0.006	12.88	2.899	0.004	14.37
32	0.414	0.011	13.89	1.051	0.010	22.41	1.484	0.006	27.99
64	0.217	0.030	23.89	0.528	0.022	43.24	0.777	0.017	52.52

Using this methodology, the three Real cases were run again. Table 2.4 presents the performance results of this "load-balanced" mesh refinement step. Compared to the results in Table 2.3, the parallel speedups are now much higher. In fact, the speedups for Real_2R consistently beats the corresponding speedups for

RANDOM_2R, while REAL_3R outperforms RANDOM_3R when more than eight processors are used. Even though the RANDOM cases are expected to behave somewhat ideally, these results show that explicit load balancing can do better. An efficiency of 82% is attained for REAL_3R on 64 processors, thereby demonstrating that mesh adaption can deliver excellent speedups if the marked edges are well-distributed among the processors. Communication requires a larger fraction of the total time for the cases in Table 2.4 than for the cases in Table 2.3. This is because the mesh refinement work is distributed among more processors after load balancing. However, communication times are still relatively small, requiring less than 4% of the total time for all runs except for REAL_1R on 64 processors.

Table 2.5: Quality of Load Balance Before and After Mesh Refinement

	RANDOM_3R		NLB REAL_3R		LB REAL_3R	
P	Before	After	Before	After	Before	After
1	1.000	1.000	1.000	1.000	1.000	1.000
2	1.000	1.016	1.000	1.556	1.406	1.000
4	1.000	1.033	1.000	2.188	1.948	1.000
8	1.000	1.085	1.000	6.347	2.654	1.000
16	1.000	1.167	1.000	5.591	4.025	1.000
32	1.001	1.226	1.001	7.987	4.212	1.000
64	1.005	1.506	1.005	8.034	6.709	1.004

The effect of load balancing the refined mesh before performing the actual subdivision can be seen more directly from the results presented in Table 2.5 for RANDOM_3R and REAL_3R. The quality of load balance is defined as the ratio of the number of elements on the most heavily-loaded processor to the number of elements on the most lightly-loaded processor. For the RANDOM_3R strategy, the mesh was refined without any load balancing. Two different sets of results are presented for REAL_3R: one without load balancing (NLB) and the other using the technique of load-balanced mesh refinement (LB). Notice that the quality of load balance before refinement is excellent, and identical, for both RANDOM_3R and NLB REAL_3R because the initial mesh is partitioned using PMeTiS [40]. However, after mesh refinement, the load imbalance is severe, particularly for NLB REAL_3R. The load imbalance is not too bad for RANDOM_3R since edges are randomly marked for refinement. This is reflected by the difference in the speedup values in Tables 2.2 and 2.3. For LB REAL_3R, the initial mesh is repartitioned after edge marking is complete. This imbalances the load before refinement, but generates excellently balanced partitions after subdivision is complete. It also improves the speedup values significantly.

2.4.2 Coarsening Phase

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

Table 2.6: Performance of Mesh Coarsening

	RANDOM_2C				REAL_2C			
P	Comp Time	Comm Time	Comm Time	Total Speedup	Comp Time	Comm Time	Comm Time	Total Speedup
1	3.619	2.364	0.001	1.00	3.989	2.246	0.000	1.00
2	1.832	1.352	0.002	1.88	2.026	1.283	0.000	1.88
4	0.963	0.782	0.004	3.42	1.066	0.854	0.000	3.25
8	0.572	0.498	0.005	5.57	0.600	0.498	0.000	5.68
16	0.303	0.287	0.008	10.01	0.334	0.279	0.000	10.17
32	0.170	0.170	0.013	16.95	0.167	0.161	0.000	19.01
64	0.070	0.098	0.024	31.17	0.093	0.097	0.000	32.82

Timings and parallel speedup for the RANDOM_2C and the REAL_2C coarsening strategies are presented in Table 2.6. Note that the follow-up mesh refinement times are not included. This was done in order to demonstrate the parallel performance of the modules that are only required during the coarsening phase. The computation time in Table 2.6 is the time required to mark edges for coarsening.

Notice that the communication time is generally negligible for RANDOM_2C and identically zero for REAL_2C. No communication was required for REAL_2C to decide which edges to coarsen. The amount of communication needed during the coarsening phase depends both on the problem and the nature of the coarsening strategy; however, the situation can never be worse than the corresponding RANDOM case. The cleanup time, on the other hand, is always a significant fraction of the total time. The cleanup time decreases as more and more processors are used due to the reduction in the local mesh size for each individual partition; however, since it depends on the fraction of shared objects, performance deteriorates as the problem size is over-saturated by processors. For instance, even though the total efficiency is about 50% for 64 processors for the results in Table 2.6, the efficiency when considering only the cleanup times is barely 37%.

2.4.3 Initialization and Finalization Phases

Table 2.7. Performance of Initialization and Finalization Steps for REAL_1R Strategy

	Initialization			Finalization		
P	Comp Time	Bcast Time	Total Speedup	Comp Time	Gather Time	Total Speedup
1	6.098	0.344	1.00	11.380	1.227	1.00
2	3.315	0.677	1.61	8.309	1.154	1.33
4	1.807	1.199	2.14	4.410	1.136	2.27
8	1.074	0.857	3.34	3.340	1.169	2.80
16	0.622	1.022	3.92	1.973	1.202	3.97
32	0.378	1.253	3.95	1.125	1.357	5.08
64	0.330	1.605	3.33	0.652	1.497	5.87

Recall from Fig. 1.1 that unlike the execution phase where the actual adaption is performed, it is not critical for the initialization and finalization procedures to be very efficient since they are used rarely (or only once) during a flow computation. Table 2.7 presents the results for these two phases for the REAL_1R strategy. The initialization step is thus performed on the starting mesh consisting of 60,968 elements, while the finalization phase is for the refined mesh consisting of 82,489 elements. It is apparent from the timings that the performance bottleneck for the two steps are the global broadcast (one-to-all) and gather (all-to-one) communication patterns, respectively. These times generally increase with the number of processors so a speedup cannot be expected. However, the computational sections of these procedures do show favorable speedups of 18.5X and 17.5X on 64 processors. In any case, the overall run times of these routines are acceptable for our purposes. Note that the broadcast and gather times are non-zero even for a single processor because the current implementation uses a host to perform the data I/O. The number of processors shown in Table 2.7 indicates those that are actually performing the mesh adaption.

CHAPTER 3

DYNAMIC LOAD BALANCING

In this chapter, 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 [10, 11, 52, 53, 63] 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. An SP2 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. Results show that our parallel balancing strategy for adaptive unstructured meshes will remain viable on large numbers of processors as none of the individual modules will be a bottleneck.

3.1 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.

3.2 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 $(W_{\mathrm{max}} - W_{\mathrm{avg}})$. This is an exact measure of the load imbalance. The mesh is repartitioned if the imbalance factor $W_{\mathrm{max}}/W_{\mathrm{avg}}$ is unacceptable.

3.3 Parallel Mesh Repartitioning

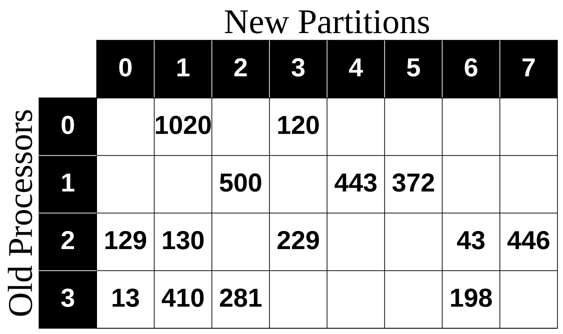
If the preliminary evaluation step determines that the dual graph with a new weight distribution is unbalanced, the mesh needs to be repartitioned. Note that repartitioning is always performed on the initial dual graph with the weights of the vertices and edges adjusted to reflect a mesh adaption step. 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.

For the test cases in this chapter PMeTiS [41] was used as the repartitioner. PMeTiS 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 PMeTiS 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, can use 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. Extensive analysis of several other repartitioning strategies are presented in Chapter 4.

3.4 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 3.8.



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 3.1. An example of a similarity matrix M for $P = 4$ and $F = 2$. Only the non-zero entries are shown.

The first step toward processor reassignment is to compute a similarity measure M 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 Mi, 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 $F = 2$ is shown in Fig. 3.1. Only the non-zero entries are shown.

3.5 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 three general metrics: `TotalV` and `MaxV`, and `MaxSR` which model the remapping cost on most multiprocessor systems. `TotalV` minimizes the total volume of data moved among all processors, `MaxV` minimizes the maximum flow of data to **or** from any single processor, while `MaxSR` minimizes sum of the maximum flow of data to **and** from any processor. A greedy heuristic algorithm is also presented.

3.5.1 TotalV metric

The `TotalV` metric 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. To minimize `TotalV`, each processor i must be assigned F partitions

j_{i_f}, $f = 1, 2, \ldots, F$, such that the objective function

$$\mathcal{F} = \sum_{i=1}^{P} \sum_{f=1}^{F} M_{i j_{i\_f}} \tag{3.1}$$

is maximized subject to the constraint

$$j_{i\_r} \neq j_{k\_s}, \text{ for } i \neq k \text{ or } r \neq s; \quad i, k = 1, 2, \ldots, P; \quad r, s = 1, 2, \ldots, F.$$

We can optimally solve this by mapping it to a network flow optimization problem described as follows. Let $G = (V, E)$ be an undirected graph. G is bipartite if V can be partitioned into two sets A and B such that every edge has one vertex in A and the other vertex in B. A matching is a subset of edges, no two of which share a common vertex. A maximum-cardinality matching is one that contains as many edges as possible. If G has a real-valued cost on each edge, we can consider the problem of finding a maximum-cardinality matching whose total edge cost is maximized. We refer to this as the maximally weighted bipartite graph (MWBG) problem (also known as the assignment problem).

When $F = 1$, optimally solving the `TotalV` metric trivially reduces to MWBG, where V consists of P processors and P partitions in each set. An edge of weight M_{ij} 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 MWBG 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 for the `TotalV` metric and the corresponding processor assignment of an example similarity matrix is shown in Fig. 3.2(a).

The fastest MWBG algorithm can compute a matching in $O(|V|^2 \log |V| + |V||E|)$ time [27], or in $O(|V|^{1/2}|E| \log(|V|C))$ time if all edge costs are integers of absolute value at most C [28]. We have implemented the optimal algorithm with a runtime of $O(|V|^3)$. Since M is generally dense, $|E| \approx |V|^2$, implying that we should not see a dramatic performance gain from a faster implementation.

3.5.2 MaxV metric

The metric `MaxV`, unlike `TotalV`, 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 max($\alpha\times$max(`ElemsSent`), $\beta\times$max(`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. We can solve this optimally by considering the problem of finding a maximum-cardinality matching whose maximum edge cost is minimum. We refer to this as the bottleneck maximum cardinality matching (BMCM) problem.

To find the BMCM of the graph G corresponding to the similarity matrix, we first need to transform M into a new matrix M'. Each entry M'_{ij} represents the maximum cost of sending data to or receiving data from processor i and partition j:

$$M'_{ij} = \max\Big((\alpha \sum_{y=1}^{P} M_{iy}, y \neq j), (\beta \sum_{x=1}^{P} M_{xj}, x \neq i)\Big). \tag{3.2}$$

Currently, our framework for the `MaxV` metric is restricted to $F = 1$. We have implemented the BMCM algorithm of Bhat [5] which combines a maximum cardinality matching algorithm with a binary search, and runs in $O(|V|^{1/2}|E|\log|V|)$. The fastest known BMCM algorithm, proposed by Gabow and Tarjan [29], has a runtime of $O((|V|\log|V|)^{1/2}|E|)$.

The new processor assignment for the similarity matrix in Fig. 3.2 using this approach with $\alpha = \beta = 1$ is shown in Fig. 3.2(b). Notice that the total number

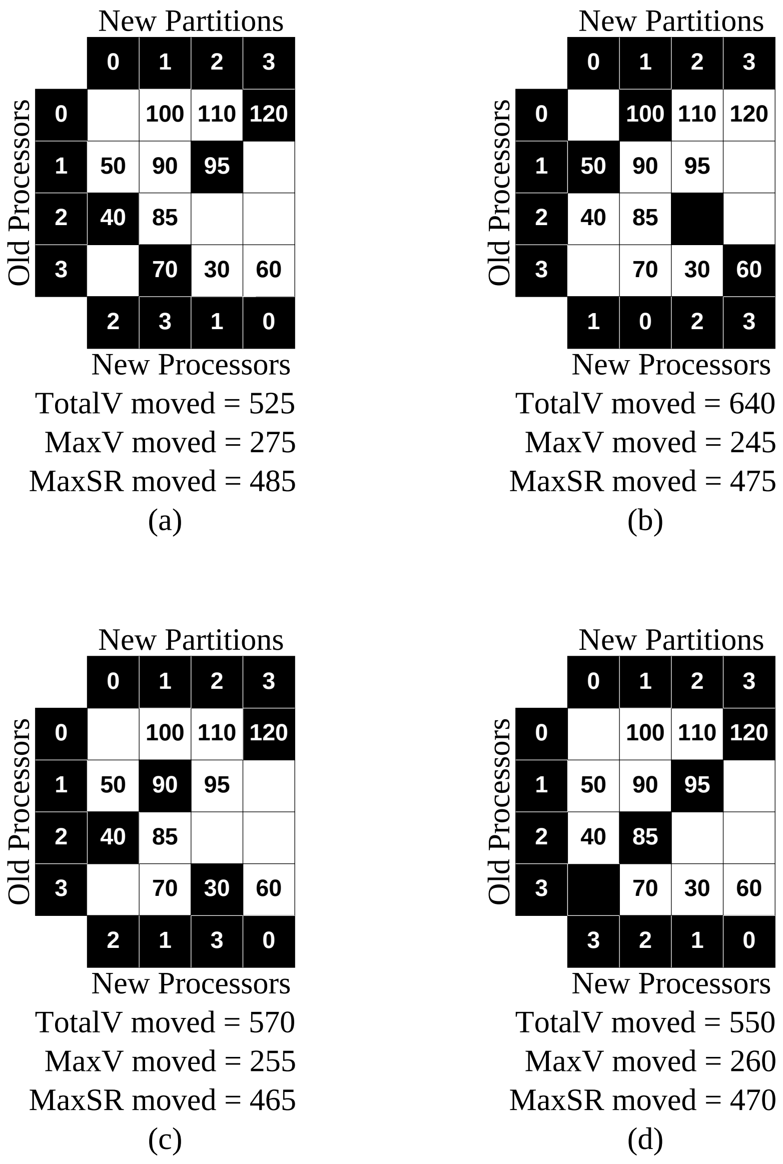


Figure 3.2. Various cost metrics of a similarity matrix M for $P = 4$ and $F = 1$ using (a) optimal MWBG algorithm, (b) optimal BMCM algorithm, (c) optimal DBMCM algorithm, and (d) our heuristic algorithm.

of elements moved in Fig. 3.2(b) is larger than the corresponding value in Fig. 3.2(a); however, the maximum number of elements moved is smaller.

3.5.3 MaxSR metric

Our third metric, `MaxSR`, is similar to `MaxV` in the sense that the overhead of the bottleneck processor is minimized during the remapping phase. `MaxSR` differs, however, in that it minimizes the sum of the heaviest data flow **from** any processor and **to** any processor, expressed as ($\alpha\times$max(`ElemsSent`) + $\beta\times$max(`ElemsRecd`)). We refer to this as the double bottleneck maximum cardinality matching (DBMCM) problem. The `MaxSR` formulation allows us to capture the computational overhead of packing and unpacking data, when these two phases are separated by a barrier synchronization. Additionally, the `MaxSR` metric may also approximate the many-to-many communication pattern of our remapping phase. Since a processor can either be sending or receiving data, the overhead of these two phases should be modeled as a sum of costs.

We have developed an algorithm for computing the minimum `MaxSR` of the graph G corresponding to our similarity matrix. We first transform M to a new matrix M''. Each entry M''_{ij} contains a pair of values (*Send,Receive*) representing the total cost of sending and receiving data, when processor i is mapped to partition j:

$$M''_{ij} = \Big\{S_{ij} = (\alpha \sum_{y=1}^{P} M_{iy}, y \neq j)\,,\, R_{ij} = (\beta \sum_{x=1}^{P} M_{xj}, x \neq i)\Big\}. \qquad (3.3)$$

Currently, our algorithm for the `MaxSR` metric is restricted to $F = 1$.

Let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be the distinct *Send* values appearing in M'', sorted in increasing order. Thus, $\sigma_i < \sigma_{i+1}$ and $k \leq P^2$. Form the bipartite graph $G_i = (V, E_i)$, where V consists of processor vertices $u = 1, 2, \ldots, P$ and partition vertices $v = 1, 2, \ldots, P$, and E_i contains edge (u, v) if $S_{uv} \leq \sigma_i$; furthermore, edge (u, v) has weight R_{uv} if it is in E_i.

For small values of i, graph G_i may not have a perfect matching. Let $i_{\min}$ be the smallest index such that $G_{i_{\min}}$ has a perfect matching. Obviously, G_i has a perfect matching for all $i \geq i_{\min}$. Solving the BMCM problem of G_i gives a matching that minimizes the maximum *Receive* edge weight. It gives a matching with `MaxSR` value at most $\sigma_i +$ `MaxV`(G_i). Define

$$\texttt{MaxSR}(i) = \min_{i_{\min} \leq j \leq i} (\sigma_j + \texttt{MaxV}(G_j)). \tag{3.4}$$

It is easy to see that `MaxSR`(k) equals the correct value of `MaxSR`. Thus, our algorithm computes `MaxSR` by solving k BMCM problems on the graphs G_i and computing the minimum value `MaxSR`(k). However, we can prematurely terminate the algorithm if there exists an $i_{\max}$ such that $\sigma_{i_{\max}+1} \geq$ `MaxSR`$(i_{\max})$, since it is then guaranteed that the `MaxSR` solution is `MaxSR`$(i_{\max})$.

Our implementation has a runtime of $O(|V|^{1/2}|E|^2 \log |V|)$ since the BMCM algorithm is called $|E|$ times in the worst case; however, it can be decreased to $O(|E|^2)$. The following is a brief sketch of this more efficient implementation.

Suppose we have constructed a matching $\mathcal{M}$ that solves the BMCM problem of G_i for $i \geq i_{\min}$. We solve the BMCM problem of G_{i+1} as follows. Initialize a working graph G to be G_{i+1} with all edges of weight greater than `MaxV`(G_i) deleted. Take the matching $\mathcal{M}$ on G, and delete all unmatched edges of weight `MaxV`(G_i). Choose an edge (u, v) of maximum weight in $\mathcal{M}$. Remove edge (u, v) from $\mathcal{M}$ and G, and search for an augmenting path from u to v in G. If no such path exists, we know that `MaxV`(G_i) =`MaxV`(G_{i+1}). If an augmenting path is found, repeat this procedure by choosing a new edge (u', v') of maximum weight in the matching and searching for an augmenting path. After some number of repetitions of this procedure, the maximum weight of a matched edge will have decreased to the desired value `MaxV`(G_{i+1}). At this point our algorithm to solve the BMCM problem of G_{i+1} will stop, since no augmenting path will be found.

This algorithm runs in total time $O(|E|^2)$. To see this, note that each search for an augmenting path uses time $O(|E|)$. The total number of such searches is $O(|E|)$. This is because a successful search for an augmenting path for edge (u, v) permanently eliminates this edge from all future graphs, so there are at most $|E|$ successful searches. Furthermore, there are at most $|E|$ unsuccessful searches, one for each value of i.

The new processor assignment for the similarity matrix in Fig. 3.2 using the DBMCM algorithm with $\alpha = \beta = 1$ is shown in Fig. 3.2(c). Notice that the `MaxSR` solution is minimized; however, the number of `TotalV` elements moved is larger than the corresponding value in Fig. 3.2(a), and more `MaxV` elements are moved than in Fig. 3.2(b). Also note that the optimal similarity matrix solution for `MaxSR` is provably no more than twice that of `MaxV`.

3.5.4 Heuristic Algorithm

We have developed a heuristic greedy algorithm that gives a suboptimal solution to the `TotalV` metric in $O(|E|)$ steps. The pseudocode for our heuristic algorithm is given in Fig. 3.3. 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 M 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 M are also used. Applying this heuristic algorithm to the similarity matrix in Fig. 3.2 generates the new processor assignment shown in Fig. 3.2(d). We show that 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 `TotalV` assignment. Additionally, experimental results in Section 3.8 demonstrate that our heuristic quickly finds high quality solutions for all three metrics.

```
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 M[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 3.3. Pseudocode for our heuristic algorithm for solving the processor reassignment problem.

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

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

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

<u>Case 1.</u> $M^{n+1}_{n+1,n+1}$ is contained in the optimal solution.

This means $\mathtt{Opt}^{n+1} = \mathtt{Opt}^n + M^{n+1}_{n+1,n+1}$. Thus, $2\,\mathtt{Heu}^{n+1} \geq \mathtt{Opt}^{n+1} + M^{n+1}_{n+1,n+1}$, which implies $2\,\mathtt{Heu}^{n+1} \geq \mathtt{Opt}^{n+1}$. $\square$

<u>Case 2.</u> $M^{n+1}_{n+1,n+1}$ is not contained in the optimal solution.

Without loss of generality, assume that $M^{n+1}_{n,n+1}$ and $M^{n+1}_{n+1,n}$ are contained in the optimal solution. This means $\texttt{Opt}^{n+1} = \texttt{Opt}^{n-1} + M^{n+1}_{n,n+1} + M^{n+1}_{n+1,n}$. By definition of $M^{n+1}_{n+1,n+1}$, we get $\texttt{Opt}^{n+1} \leq \texttt{Opt}^{n-1} + 2\, M^{n+1}_{n+1,n+1}$. Since $\texttt{Opt}^{n} \geq \texttt{Opt}^{n-1}$, we have $\texttt{Opt}^{n+1} \leq \texttt{Opt}^{n} + 2\, M^{n+1}_{n+1,n+1}$. Therefore, $2\,\texttt{Heu}^{n+1} \geq \texttt{Opt}^{n+1}$. $\square$

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 $\sum\sum M_{i,j} - \mathcal{F}$. We need to show that $\sum\sum M^{n}_{i,j} - \texttt{Heu}^{n} \leq 2\,(\sum\sum M^{n}_{i,j} - \texttt{Opt}^{n})$; that is, $\sum\sum M^{n}_{i,j} - 2\,\texttt{Opt}^{n} + \texttt{Heu}^{n} \geq 0$.

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, $\sum\sum M^{n}_{i,j} \geq \texttt{Opt}^{n} + \texttt{Heu}^{n} - \texttt{Int}^{n}$. This implies $\sum\sum M^{n}_{i,j} - 2\,\texttt{Opt}^{n} + \texttt{Heu}^{n} \geq 2\,\texttt{Heu}^{n} - \texttt{Opt}^{n} - \texttt{Int}^{n}$. 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 $M^{k} \subseteq M^{n}$. $\square$

Recall that `TotalV` does not consider the execution times of bottleneck processors while `MaxV` and `MaxSR` ignore bandwidth contention. A quantitative comparison of all three metrics is presented in Section 3.8. In general, the objective function may need to use a combination of metrics to effectively incorporate all related costs.

3.6 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:

$$\alpha \times \max(\texttt{ElemsSent}) + \beta \times \max(\texttt{ElemsRecd}) + \delta, \tag{3.5}$$

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 [19], LogGP [1], and BSP [66]. All three models make the following assumptions which hold true for most current architectures: 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 in this chapter were performed on the wide-node IBM SP2 located at NASA Ames Research Center. Portability onto the Origin2000 is addressed in Chapter 4. The SP2 consists of RS6000/590 processors, which 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 over head without the need to model multiple hops for individual messages. We approximate our communication cost for the SP2 as:

$$g \times \max(\texttt{ElemsSent}) + g \times \max(\texttt{ElemsRecd}) + l, \tag{3.6}$$

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:

$$\gamma \times \texttt{MaxSR} + O, \tag{3.7}$$

where $\texttt{MaxSR} = \max(\texttt{ElemsSent}) + \max(\texttt{ElemsRecd})$, $\gamma = \alpha + g$, and $O = \delta + l$. Eqn. 3.7 demonstrates precisely why we need to model the `MaxSR` metric when performing processor reassignment. By minimizing `MaxSR` we can guarantee a reduction in the computational overhead of our remapping algorithm. Since the computational workload is architecture independent, we are effectively solving two load balancing problems partitioned by a collective communication. Additionally, by reducing `MaxSR` we can achieve a savings in communication overhead on many bandwidth rich systems. Most modern architectures are restricted to a single port model, where each processor can either be sending or receiving a single message. The many-to-many communication pattern of remapping can therefore be approximated as a load balance problem, represented by `MaxSR`.

In order to compute the slope and intercept of the linear function in Eqn. 3.7, 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.

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 $T_{\text{iter}} N_{\text{adapt}} (W_{\text{max}}^{\text{old}} - W_{\text{max}}^{\text{new}})$, 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 $W_{\text{max}}^{\text{old}}$ and $W_{\text{max}}^{\text{new}}$ are the sum of the w_{comp} on the most heavily-loaded processor for the old and new partitioning, 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.

3.7 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 [12]. 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. An extensive analysis of the mesh adaption procedure is presented in Chapter 2.

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. 1.1).

As described in Section 2.4, 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:

$$T_{\text{iter}} N_{\text{adapt}} (W_{\max}^{\text{old}} - W_{\max}^{\text{new}}) + T_{\text{refine}} \left(\frac{W_{\max}^{\text{new}}}{W_{\max}^{\text{old}}} - 1 \right) > \gamma \times \texttt{MaxSR} + O, \tag{3.8}$$

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

3.8 Experimental results

PLUM was originally 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 chapter. A portability analysis of PLUM on the SGI/Cray Origin2000 is presented in Chapter 4.

Table 3.1: Grid sizes for the three different refinement strategies

	Vertices	Elements	Edges
Initial Mesh	13,967	60,968	78,343
Real_1	17,880	82,489	104,209
Real_2	39,332	201,780	247,115
Real_3	61,161	321,841	391,233

The computational mesh used for the experiments in this chapter is the one used to simulate the acoustics wind-tunnel test of Purcell [54]. In the first set of experiments, only one level of adaption is performed with varying fractions of the mesh in Fig. 2.7 being targeted for refinement. These cases, denoted as Real_1R, Real_2R, and Real_3R, were used during the parallel mesh adaption analysis of Sec. 2.4. Recall that for these strategies, edges are targeted for subdivision based on an error indicator [52] calculated directly from the flow solution. For clarity, Table 3.1 lists the grid sizes for this single level of refinement for each of the three cases. Note that the same information can be derived from Table 2.1

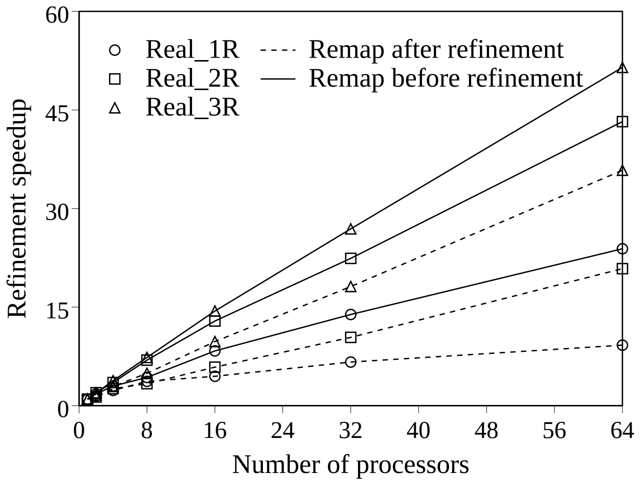


Figure 3.4. Speedup of the 3D_TAG parallel mesh adaption code when data is remapped either after or before mesh refinement.

Figure 3.4 illustrates the parallel speedup curves for each of the three edge-marking strategies, previously presented in Tables 2.3 and 2.4. Two sets of results are presented: one when data remapping is performed after mesh refinement, and the other when remapping is performed before refinement. An extensive analysis of this data is presented in Section 2.4.

Figure 3.5 shows the remapping time for each of the three cases. As in Fig. 3.4, 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_3R 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. 3.4 and 3.5 demonstrate that our methodology within PLUM is effective in significantly reducing the data remapping time and improving the parallel performance of mesh refinement.

Figure 3.6 compares the execution times and the amount of data movement for the REAL_2R strategy when using the optimal and heuristic MWBG processor

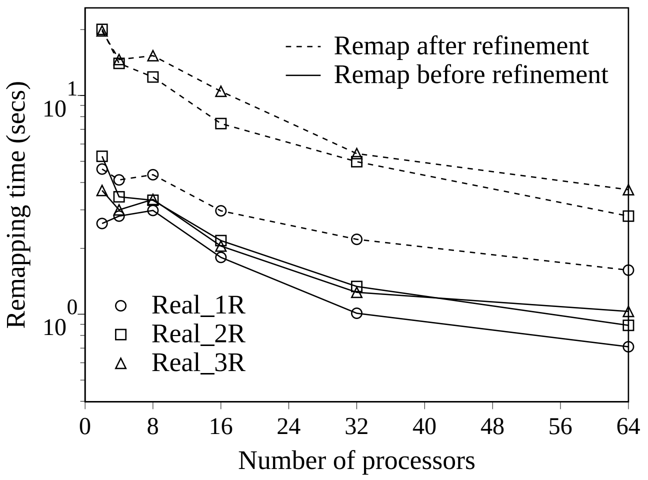
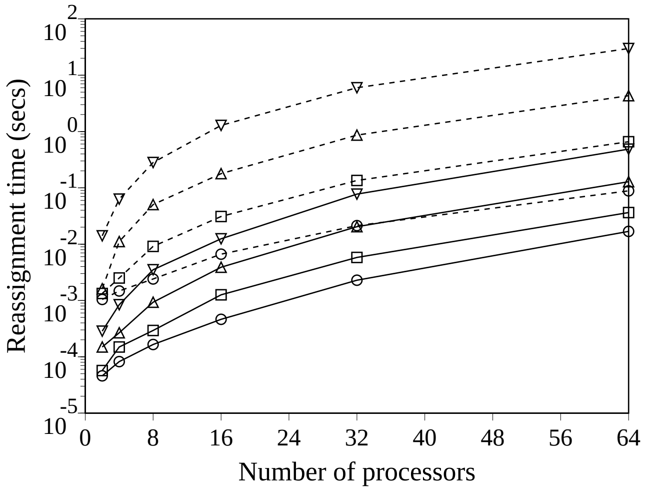


Figure 3.5. Remapping times within PLUM when data is remapped either after or before mesh refinement.

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 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. 3.6 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 3.2 presents a comparison of our five different processor reassignment strategies in terms of processor reassignment time and the amount of data movement. Results are shown for the REAL_2R strategy on the SP2 with $F = 1$. The first row shows the default assignment generated by the PMeTiS [41] partitioner, while the remaining strategies refer to our reassignment algorithms described in Section 3.5.

The PMeTiS case does not require any explicit processor reassignment since



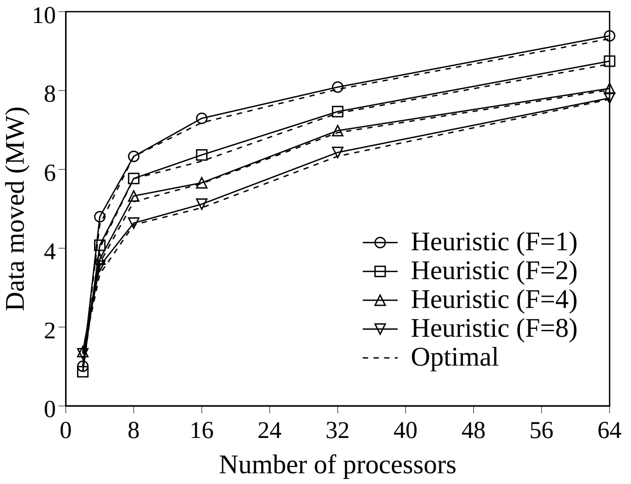


Figure 3.6. Comparison of the optimal and heuristic MWBG remappers in terms of the execution time (top) and the volume of data movement (bottom) for the Real_2R strategy.

Table 3.2. Comparison of five processor reassignment algorithms for the Real_2R case on the SP2 with $F = 1$.

	P = 8				P = 16			
Algthm.	`TotalV` Metric	`MaxV` Metric	`MaxSR` Metric	Reass. Time	`TotalV` Metric	`MaxV` Metric	`MaxSR` Metric	Reass. Time
PMeTiS	42680	9597	13359	0.0000	53242	8012	11222	0.0000
Heuristic	30071	8169	11167	0.0002	36520	7131	9294	0.0005
MWBG	30071	8169	11162	0.0013	35096	7131	9230	0.0045
BMCM	35506	8169	11512	0.0019	50488	7131	9377	0.0070
DBMCM	33862	8250	11010	0.0167	53012	7134	9123	0.0614

	P = 32				P = 64			
Algthm.	`TotalV` Metric	`MaxV` Metric	`MaxSR` Metric	Reass. Time	`TotalV` Metric	`MaxV` Metric	`MaxSR` Metric	Reass. Time
PMeTiS	58297	5067	7467	0.0000	67439	2667	4452	0.0000
Heuristic	35032	4410	5809	0.0017	38283	2261	3123	0.0088
MWBG	34738	4410	5822	0.0177	38059	2261	3142	0.0650
BMCM	49611	4410	5944	0.0323	52837	2261	3282	0.1327
DBMCM	50270	4414	5733	0.0921	54896	2261	3121	1.2515

we choose the default partition-to-processor mapping given by the partitioner. However, it shows extremely poor performance for all three metrics. This is expected since PMeTiS is a global partitioner that does not attempt to minimize the remapping overhead. An extensive comparison of PMeTiS with other global and diffusive partitioners is given in Section 4.2.1

The execution times of the other four algorithms increase with the number of processors because the growth in the size of similarity matrix; however, the heuristic time for 64 processors is still very small and acceptable. The total volume of data movement is obviously smallest for the MWBG algorithm since it optimally solves for the `TotalV` metric. In the optimal BMCM method, the maximum of the number of elements sent or received is explicitly minimized, but almost all the other algorithmic solutions give the identical result. 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. In our helicoptor rotor experiment, small regions of the domain incur a dramatic increase in grid points between refinement levels. These newly refined regions must shift a large number of elements onto other processors in order to achieve a balanced load distribution. Therefore, a similar `MaxV` solution should be obtained by any reasonable reassignment algorithm.

The DBMCM algorithm optimally reduces `MaxSR` metric, but achieves no more than a 5% improvement over the other algorithms. Nonetheless, since we believe that the `MaxSR` metric can closely approximate the remapping cost on many architecture, computing its optimal solution can provide useful information. Notice that the minimum `TotalV` increases slightly as P grows from 8 to 64, while the `MaxSR` is dramatically reduced by over 70%. This trend continues as the number of processors increase. These results indicates that the our load balancing algorithm will remain viable on a large number of processor, since the per processor work load decreases as P increases.

Finally, observe that the heuristic algorithm does an excellent job in minimizing all three cost metrics, in a trivial amount of time. Although theoretical bounds have only been established for the `TotalV` metric, empirical evidence indicates that the heuristic algorithm closely approximates both `MaxV` and `MaxSR`. It was therefore used to perform the processor reassignment for all the experiments reported in this paper.

Figure 3.7 shows how the execution time is spent during the refinement and the subsequent 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 3.2 for all the three cases. The repartitioning curves, using PMeTiS [41], 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_2R, the PMeTiS 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_2R.

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 $\mathcal{G}N$ elements, $1 \leq \mathcal{G} \leq 8$ for our refinement procedure. If the workload were balanced, each processor would have $\mathcal{G}N/P$ 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 $\mathcal{G}N-(P-1)N/P$ elements. Thus, the maximum improvement due to load balancing for a **single** refinement step would be:

$$\frac{1}{\mathcal{G}} \min \left(8, P(\mathcal{G}-1)+1\right) \tag{3.9}$$

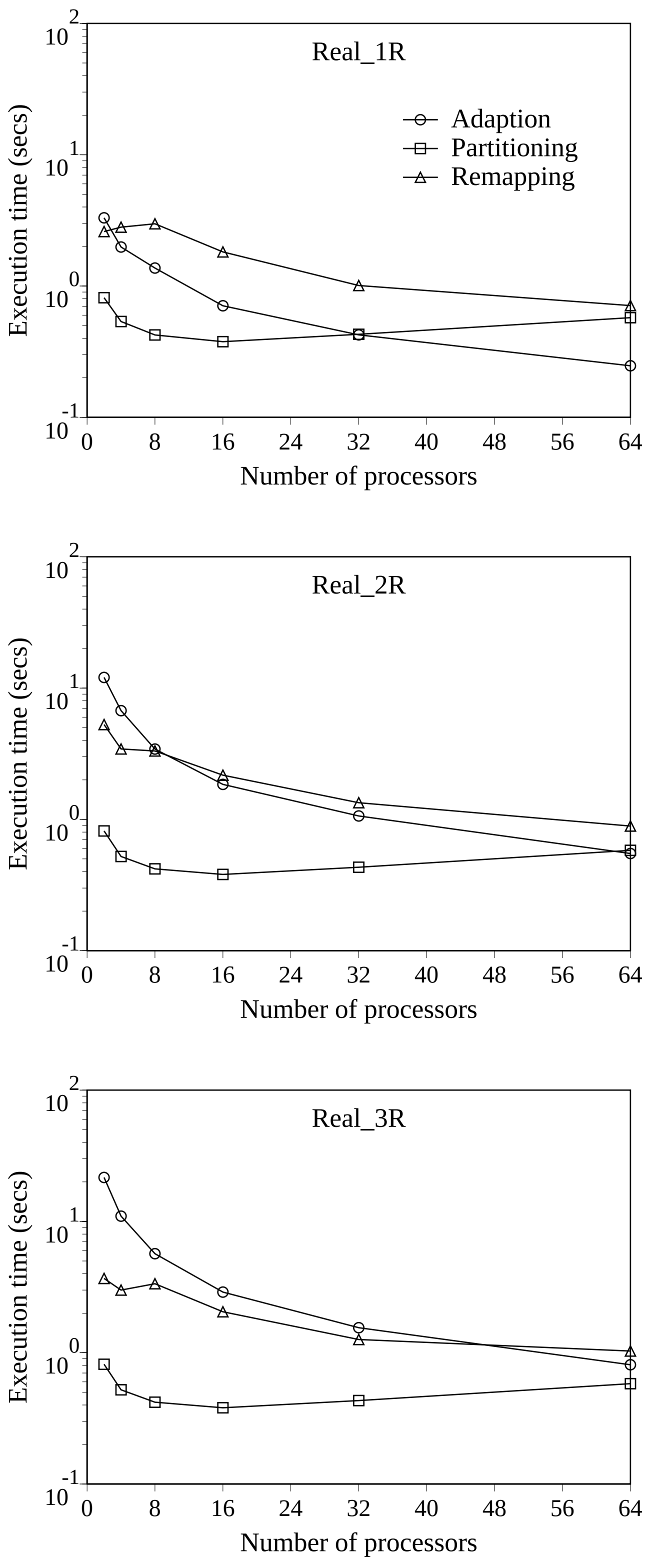


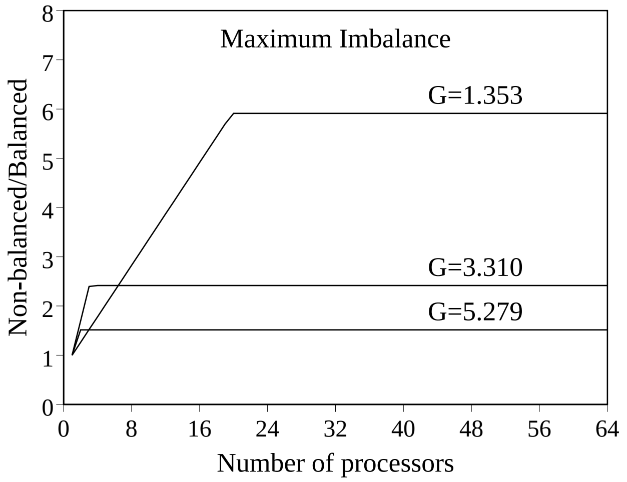
Figure 3.7. Anatomy of execution times for the Real_1R, Real_2R, and Real_3R refinement strategies.

The maximum impact of load balancing for the three strategies are shown in the top half of Fig. 3.8. The mesh growth factor $\mathcal{G}$ is 1.35 for the REAL_1R case, giving a maximum improvement of 5.91 with load balancing when $P \geq 20$. The value of $\mathcal{G}$ is 3.31 and 5.28 for REAL_2R and REAL_3R, 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 $\mathcal{G} = 1$ or $\mathcal{G} = 8$. Notice that maximum imbalance is attained faster as $\mathcal{G}$ increases; however, the magnitude of the maximum imbalance gradually decreases. The actual impact of load balancing is shown in the bottom half of Fig. 3.8. 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_1R, REAL_2R, and REAL_3R, 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_3R. The REAL_1R and REAL_2R strategies would also attain their respective maxima if more processors were used. It is important to realize that the results shown in Fig. 3.8 are for a single refinement step. With repeated refinement, the gains realized with load balancing may be even more significant.

Table 3.3. 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

In the second set of experiments, a total of three levels of adaption are performed in sequence on the mesh shown in Fig. 2.7. Table 3.3 shows the size of the computational mesh after each adaption step. Notice that the final mesh is more



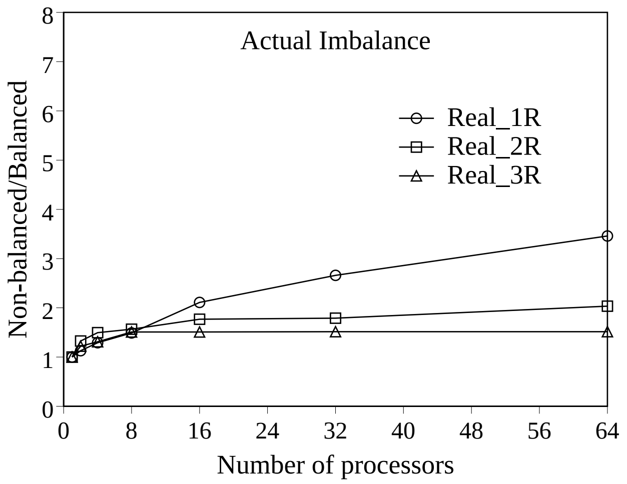


Figure 3.8. Maximum (top) and actual (bottom) impact of load balancing on flow solver execution times for different mesh growth factors $\mathcal{G}$.

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. 3.9. 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.

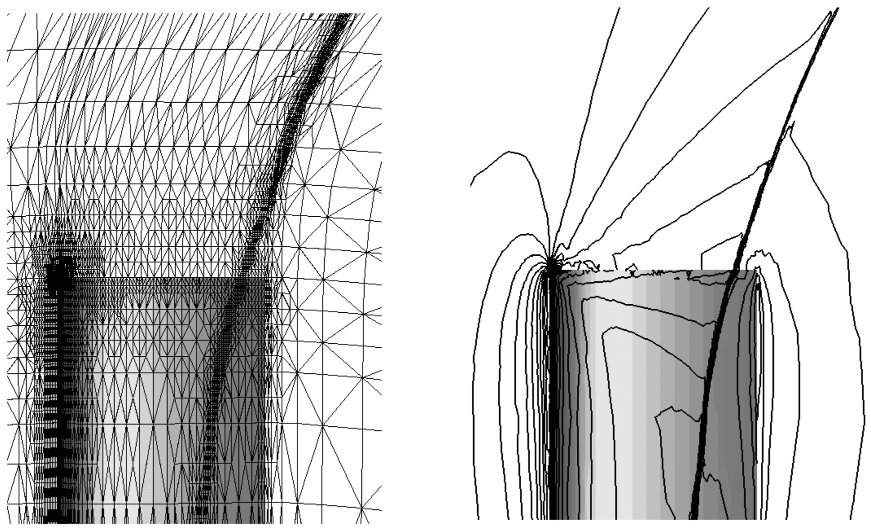


Figure 3.9. Final adapted mesh and computed pressure contours in the plane of the helicopter rotor.

Figure 3.10 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 PMeTiS [41], are almost identical to those shown in Fig. 3.7. 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

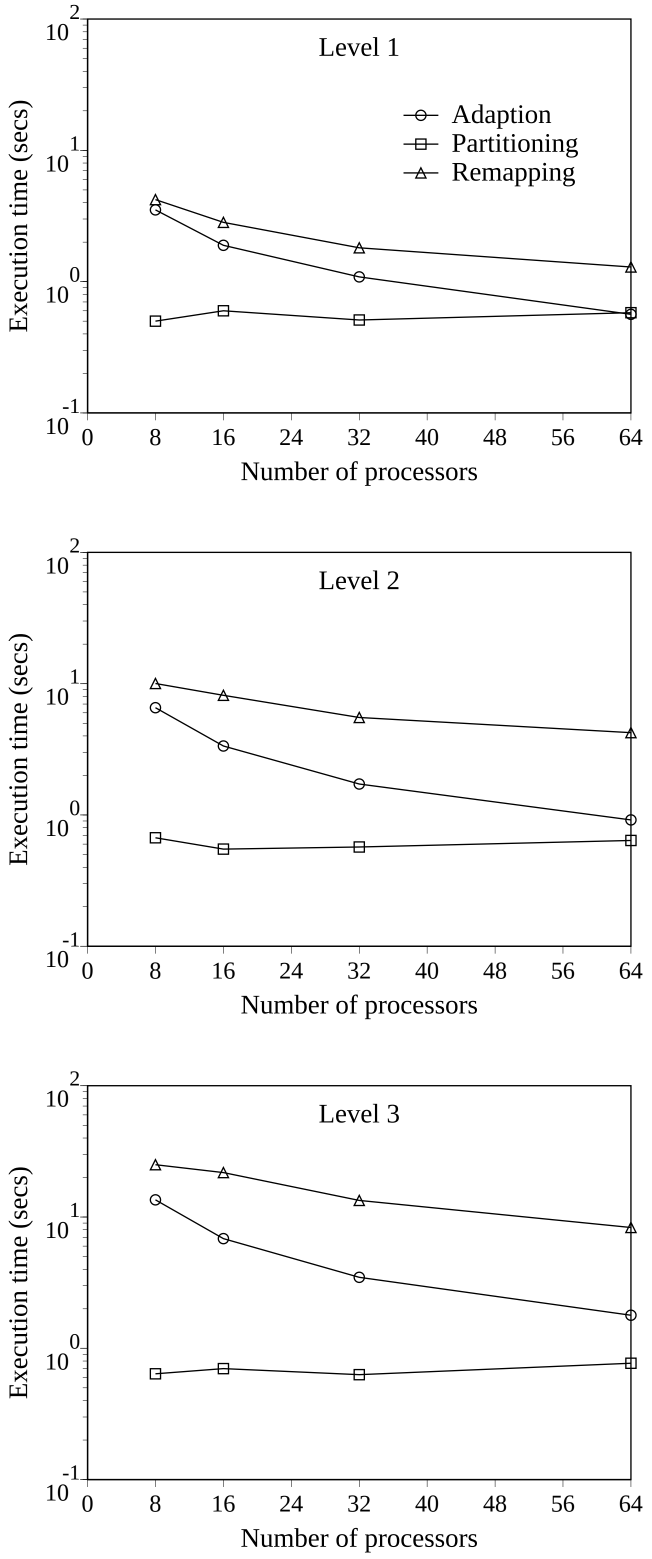
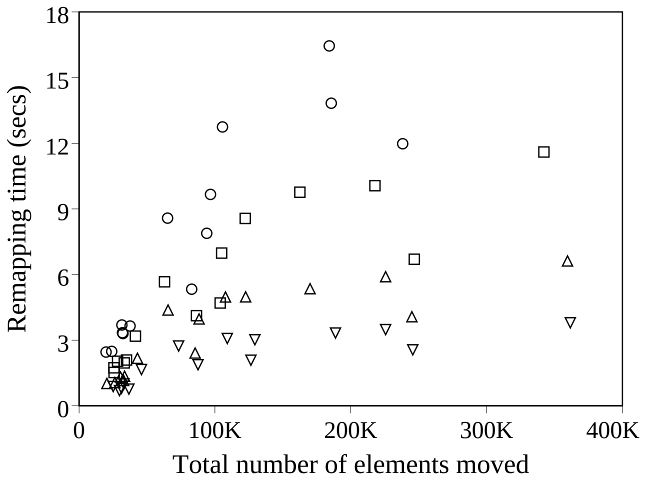


Figure 3.10: Anatomy of execution times for the three levels of adaption.

about four times the adaption time on 64 processors. This is not unexpected since remapping is considered 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.

The third set of experiments are performed to compute the slope γ and the intercept O of our SP2 redistribution cost model derived in Eqn. 3.7. 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 $\mathcal{G}$ (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. 3.11. 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 on the SP2 reducing the bottleneck, rather than the aggregate, overhead guarantees a reduction in the redistribution time.



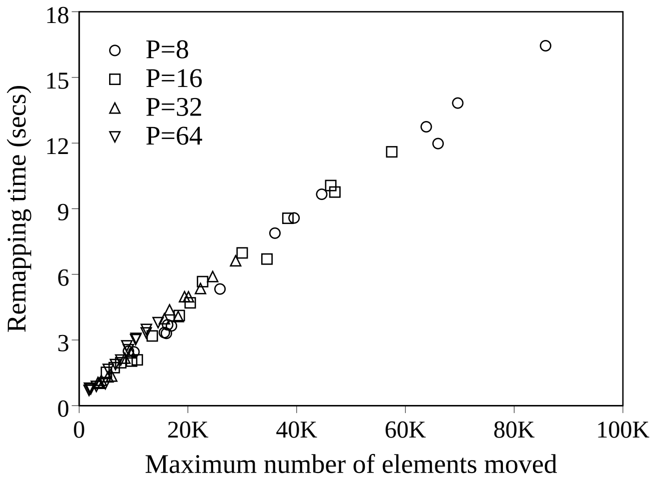


Figure 3.11. Remapping time as a function of the `TotalV` (top) and the `MaxSR` (bottom) metrics.

CHAPTER 4

PORTABILITY AND REPARTITIONING ANALYSIS

In this chapter, several experimental results verify the effectiveness of PLUM on sequences dynamically adapted unstructured grids. We examine portability by comparing results between the distributed-memory system of the IBM SP2, and the Scalable Shared-memory MultiProcessing (S2MP) architecture of the SGI/Cray Origin2000. Additionally, we evaluate the performance of five state-of-the-art partitioning algorithms that can be used within PLUM. Results indicate that for certain classes of unsteady adaption, globally repartitioning the computational mesh produces higher quality results than diffusive repartitioning schemes. We also demonstrate that a coarse starting mesh produces high quality load balancing, at a fraction of the cost required for a fine initial mesh. Finally, we show that the data redistribution overhead can be significantly reduced by applying our heuristic processor reassignment algorithm to the default partition-to-processor mapping given by partitioners.

4.1 Helicopter rotor test case

We present a portability analysis by comparing the SP2 results from section 3.8 with Origin2000 performance. The tetrahedral mesh described in Fig. 2.7 is targeted for one level of refinement, based on the three different marking strategies REAL_1R, REAL_2R, and REAL_3R (cf. Table 3.1).

All experiments were performed on a wide-node IBM SP2 and a SGI/Cray Origin2000. Note that no architecture-specific optimizations were used to obtain the performance results reported in this chapter. The SP2 is located in the Numerical

Aerospace Simulation division at NASA Ames Research Center. An overview of its architecture was presented in Section 3.6.

The Origin2000 used in these experiments is a 32-processor R10000 system, located at NCSA, University of Illinois. The Origin2000 is the first commercially-available 64-bit cache-coherent nonuniform memory access (CC-NUMA) system. A small high performance switch connects two CPUs, memory, and I/O. This module, called a node, is then connected to other nodes in a hypercube fashion. An advantage of this interconnection system is that additional nodes and switches can be added to create larger systems that scale with the number of processors. Unfortunately, this configuration causes an increase in complexity when predicting communication overhead, since an accurate cost model must consider the number of module hops, if any, between communicating processors.

4.1.1 PLUM on the Origin2000

Figure 4.1 illustrates the parallel speedup for each of the three edge-marking strategies on the Origin2000. Similar to the SP2 experiment (cf. Fig. 3.4), two sets of results are presented: one when data remapping is performed after mesh refinement, and the other when remapping is performed before refinement. Speedup numbers on the Origin2000 are almost identical to those on the SP2. The Real_3R 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_1R, because it has the smallest refinement region and load balancing the refined mesh before actual subdivision returns the biggest benefit. The results are the best for Real_3R with data remapping before refinement, showing an efficiency of more than 87% on 32 processors of both the SP2 and the Origin2000. Extensive performance analysis of the parallel mesh adaption code on the SP2 are presented in Section 2.4.

To compare the performance on the SP2 and the Origin2000 more critically,

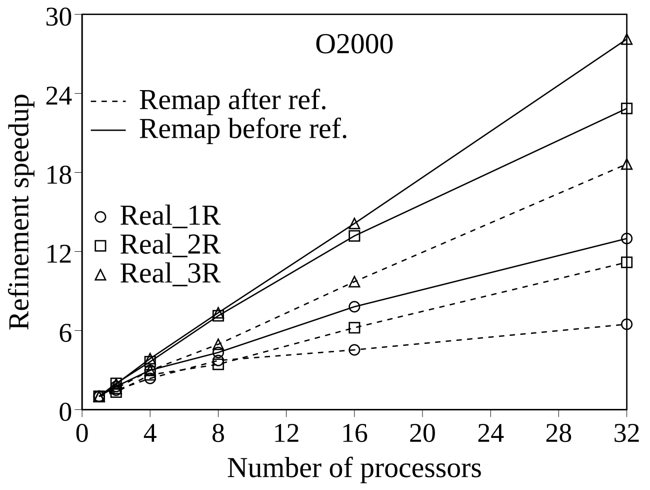


Figure 4.1. Speedup of 3D_TAG the Origin2000 when data is remapped either after or before mesh refinement.

one needs to look at the actual mesh adaption times rather than the speedup values. These results are presented in Table 4.1 for the case when data is remapped before the mesh refinement phase. Notice that the Origin2000 is consistently more than twice as fast as the SP2. One reason is the faster clock speed of the Origin2000. Another reason is that the mesh adaption code does not use the floating-point units on the SP2, thereby adversely affecting its overall performance.

Figure 4.2 shows the remapping time for each of the three cases on the Origin2000. As in the SP2 experiment (cf. Fig. 3.5), results are presented both when the data remapping is done after and before the mesh subdivision. Once again, a significant reduction in remapping time is observed when the adapted mesh is load balanced by performing data movement prior to refinement. This is because a smaller volume of data is moved, since mesh refinement occurs after redistribution. Additionally, the remapping times decrease as the number of processors is increased. This is consistent with SP2 results. As more processors share the work, each one needs to process fewer elements. The remapping times when data is moved before mesh refinement are reproduced for both systems in Table 4.2.

Table 4.1. Execution time of 3D_TAG on the SP2 and the Origin2000 when data is remapped before mesh refinement

	Real_1R		Real_2R		Real_3R	
P	SP2	O2000	SP2	O2000	SP2	O2000
1	5.902	2.507	23.780	10.468	41.702	18.307
2	3.312	1.427	12.060	5.261	21.593	9.422
4	1.981	0.839	6.734	2.880	10.977	4.736
8	1.372	0.578	3.434	1.470	5.682	2.492
16	0.708	0.321	1.846	0.794	2.903	1.296
32	0.425	0.193	1.061	0.458	1.490	0.651
64	0.247		0.550		0.794	

Perhaps the most remarkable feature of these results is the dramatic reduction in remapping times when using all 32 processors on the Origin2000. This is probably because network contention with other jobs is essentially removed when using the entire machine. One may see similar behavior on an SP2 if all the processors in a system configuration are used.

Notice that when using up to 16 processors, the remapping times on the SP2 and the Origin2000 are comparable. Recall that the remapping phase within PLUM consists of both communication (to physically move data around) and computation (to rebuild the internal and shared data structures on each processor). We cannot report these times separately as that would require introducing several barrier synchronizations. However, since the results in Table 4.1 indicate that computation is faster on the Origin2000, it is reasonable to infer that bulk communication is faster on the SP2. Additional experimentation is required to verify these claims. In any case, the results in Figs. 4.1 and 4.2 demonstrate that our methodology within PLUM is effective in significantly reducing the data remapping time and improving the parallel performance of mesh refinement.

Figure 4.3 shows how the execution time is spent during the refinement and the subsequent load balancing phases for the three different cases on the Origin2000.

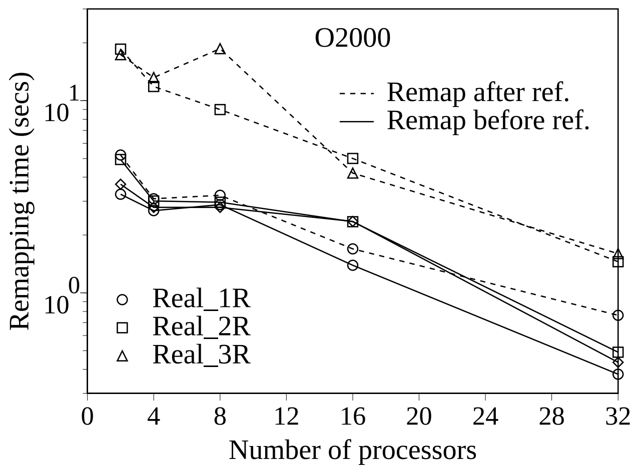


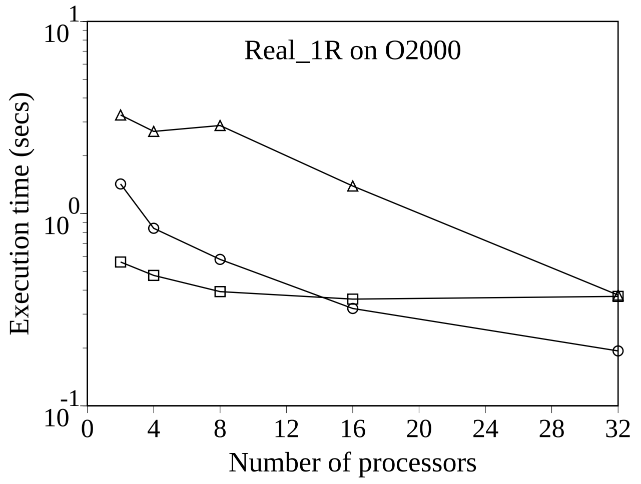
Figure 4.2. Remapping time within PLUM on the the Origin2000 when data is redistributed either after or before mesh refinement.

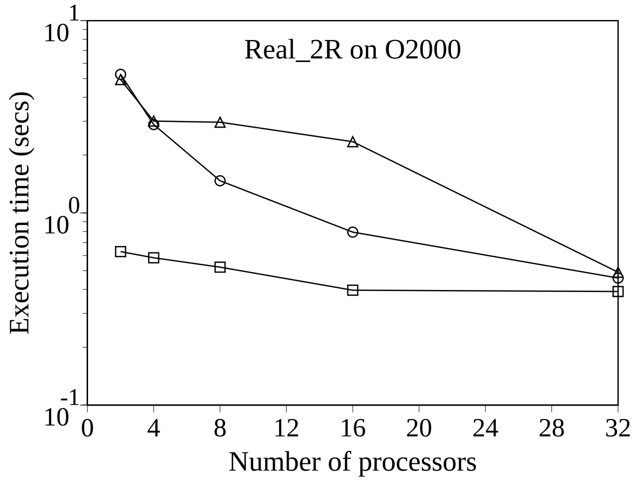
As in the SP2 results of Figure 3.7, the processor reassignment times are not shown since they are negligible compared to the other times. Note that the Origin2000 shows a qualitative behavior similar to the SP2. For all three subdivision strategies, the major components of PLUM require approximately the same amount of time when using 32 processors. These results show that PLUM can be successfully ported to different platforms without any code modifications.

4.1.2 The redistribution cost model on the Origin2000

It is important to note from the results in Fig. 4.3 and Fig. 3.7 that the refinement, repartitioning, and remapping times are generally comparable for the test mesh when using a large number of processors ($P \geq 32$). However, the remapping time will increase significantly when the mesh grows in size due to adaption. Thus, remapping is considered the bottleneck within the PLUM system. We therefore need a cost model which compares the predicted redistribution cost versus the expected computation gain of a balanced work load.

In the next set of experiments we attempt to map the SP2 redistribution cost model (cf. Sec 3.6) onto the Origin2000. Experimental data is gathered by





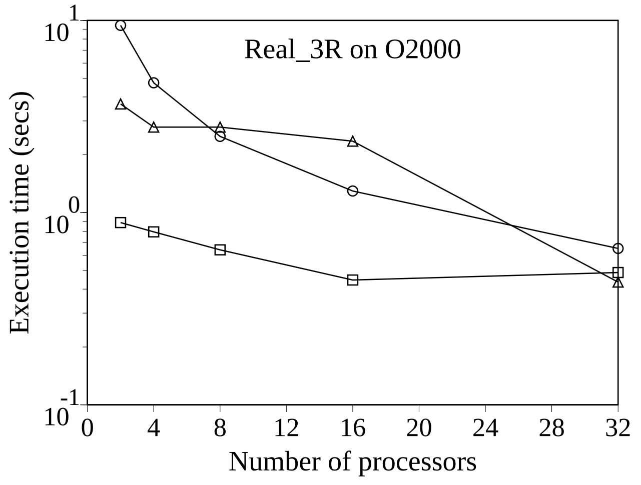


Figure 4.3. Anatomy of execution times for the Real_1R, Real_2R, and Real_3R refinement strategies on the Origin2000.

Table 4.2. Remapping time within PLUM on the SP2 and the Origin2000 when data is redistributed before mesh refinement

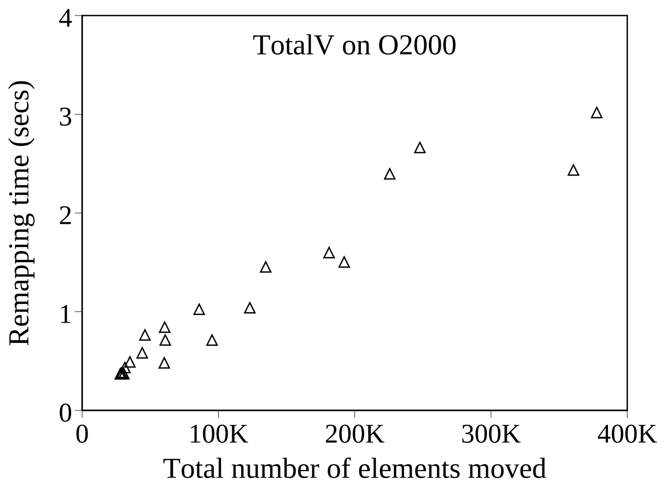
	Real_1R		Real_2R		Real_3R	
P	SP2	O2000	SP2	O2000	SP2	O2000
2	2.601	3.259	5.273	4.940	3.679	3.675
4	2.813	2.679	3.440	3.005	3.003	2.786
8	2.982	2.876	3.321	2.963	3.351	2.786
16	1.821	1.392	2.173	2.346	2.049	2.353
32	1.012	0.377	1.338	0.491	1.260	0.435
64	0.709		0.890		1.031	

running various redistribution patterns in order to compute the slope γ and the intercept O of Eqn. 3.7. The remapping times are then plotted against two metrics, `TotalV` and `MaxSR`, in Fig. 4.4. Recall Fig. 3.11 which demonstrated that our SP2 redistribution cost model successfully estimates the data remapping time. Additionally, we showed that reducing the bottleneck overhead on the SP2, results in a lower remapping overhead.

The situation is quite different on the Origin2000. Remapping times were extremely unpredictable for $P < 32$; hence, they are not shown in Fig. 4.4. Observe that, for $P = 32$, the `MaxSR` metric is not significantly better than `TotalV`. Furthermore, the `MaxSR` metric is also not as good as on the SP2. These results indicate that network contention and a complex architecture (multiple hops between processors) are probably major factors. Additional experimentation is required on the Origin2000 to develop a more reliable remapping cost model.

4.2 Unsteady simulation test case

The final set of experiments is performed to evaluate the efficacy of PLUM in an unsteady environment where the adapted region is strongly time-dependent. To achieve this goal, a simulated shock wave is propagated through the initial mesh shown at the top of Fig. 4.5. The test case is generated by refining all elements within



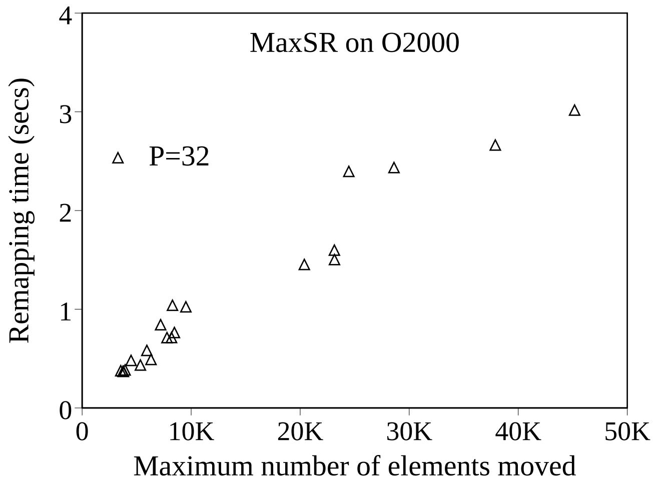


Figure 4.4: Remapping time as a function of `TotalV` and `MaxSR` on the Origin2000.

a cylindrical volume moving left to right across the domain with constant velocity, while coarsening previously-refined elements in its wake. The performance of PLUM is then measured at nine successive adaption levels. Note that because these results are derived directly from the dual graph, mesh adaption times are not reported, and remapping overheads are computed using our redistribution cost model.

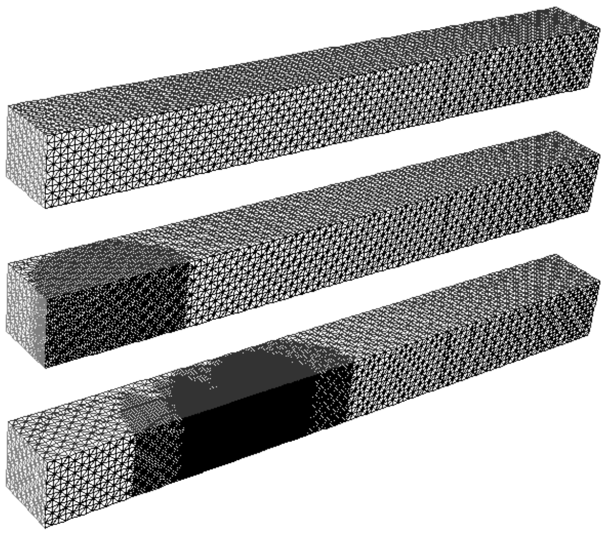


Figure 4.5. Initial and adapted meshes (after levels 1 and 5) for the simulated unsteady experiment.

Figure 4.6 shows the progression of grid sizes for the nine levels of adaption in the unsteady simulation. Both coarse and fine meshes, called Sequence_1 and Sequence_2 respectively, are used in the experiment to investigate the relationship between load balancing performance and dual graph size. The coarse initial mesh, shown in Fig. 4.5, contains 50,000 tetrahedral elements. The mesh after the first and fifth adaptions for Sequence_1 are also shown in Fig. 4.5. The initial fine mesh is eight times the size of this coarse mesh. Note that even though the size of the meshes remain fairly constant after four levels of adaption, the refinement region continues to move steadily across the domain. The growth in size due to refinement is almost exactly compensated by mesh coarsening. A third scenario, called Sequence_3, was

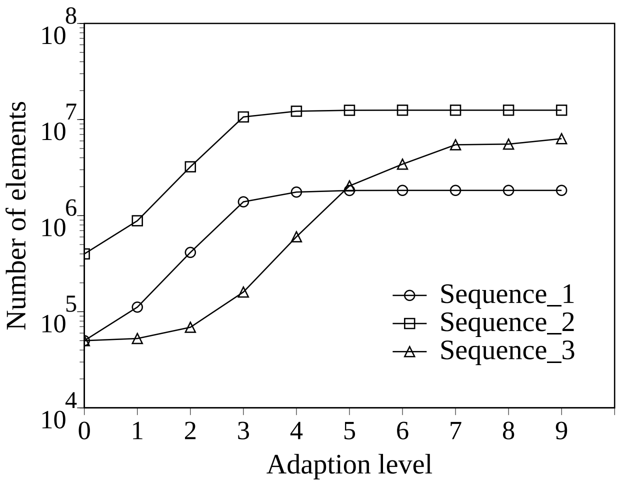


Figure 4.6. Progression of grid sizes through nine levels of adaption for the unsteady simulation.

also tested on the coarse initial mesh. This case was generated by reducing the velocity of the cylindrical volume moving across the domain. Notice that the mesh then continues to grow in size throughout the course of adaption. The final meshes after nine adaption levels contain more than 1.8, 12.5, and 6.3 million elements for Sequence_1, Sequence_2, and Sequence_3, respectively.

4.2.1 Comparison of partitioners

Recall that a good partitioning scheme is a critical component of our framework. Since PLUM can use any general partitioner, we investigate the relative performance of five parallel, state-of-the-art algorithms: PMeTiS, UAMeTiS, DAMeTiS, Jostle-MD, and Jostle-MS.

Table 4.3 presents the partitioning times for Sequence_1 using these five different partitioners briefly described in Section 1.2.4. PMeTiS is the parallel multilevel k-way partitioning scheme of Karypis and Kumar [41], UAMeTiS and DAMeTiS are multilevel undirected and directed repartitioning algorithms of Schloegel, Karypis, and Kumar [62], and Jostle-MS and Jostle-MD are multilevel-static and multilevel-dynamic configurations of the Jostle partitioner of Walshaw, Cross, and Everett [72].

Average results[1] show that UAMeTiS is the fastest among all five partitioners, while Jostle-MS is the slowest. PMeTiS is about 40% slower than UAMeTiS, but almost six times faster than Jostle-MS.

Table 4.3. Partitioning time on the SP2 for P=64 using a variety of partitioners for Sequence_1

L	PMeTiS	UAMeTiS	DAMeTiS	Jostle-MS	Jostle-MD
1	0.52	0.34	0.42	2.20	2.20
2	0.63	0.40	0.51	2.93	2.97
3	0.68	0.55	0.68	4.28	4.36
4	0.89	0.66	0.67	5.52	5.38
5	1.00	0.83	0.82	7.47	5.57
6	1.07	0.61	0.80	6.01	5.60
7	1.02	0.58	0.74	6.16	6.66
8	0.89	0.65	0.96	4.92	6.13
9	1.02	0.89	1.05	5.47	5.41
A	0.86	0.61	0.74	5.00	4.92

But partitioning time alone is not sufficient to rate the performance of a mesh partitioner; one needs to investigate the quality of load balancing as well. We define load balancing quality in two ways: the computational load imbalance factor[2] and the percentage of cut edges. These values are presented for all five partitioners both before and after they are invoked for Sequence_1 in Tables 4.4 and 4.5. PMeTiS does an excellent job of consistently reducing the load imbalance factor to within 6% of ideal (cf. Table 4.4). The Jostle partitioners are only slightly worse than PMeTiS, and turn in acceptable performances. UAMeTiS and DAMeTiS, on the other hand, show load imbalance factors larger than two. We do not know why this happens; however, a poor load imbalance factor after repartitioning at any given adaption level is one reason for a higher load imbalance factor before repartitioning at the next adaption level.

[1]The last row in Tables 4.3–4.10 is marked with an **A**. It represents the average results over all nine levels of adaption.

[2]The load imbalance factor is the ratio of the sum of the w_{comp} on the most heavily-loaded processor to the average load across all processors.

Table 4.4. Load imbalance factor before and after mesh partitioning for P=64 using a variety of partitioners for Sequence_1

	PMeTiS		UAMeTiS		DAMeTiS		Jostle-MS		Jostle-MD	
L	Bef	Aft	Bef	Aft	Bef	Aft	Bef	Aft	Bef	Aft
1	3.58	1.03	3.58	2.32	3.58	2.46	3.58	1.02	3.58	1.02
2	2.17	1.04	4.63	2.94	4.97	2.70	2.21	1.04	2.18	1.05
3	2.46	1.11	5.95	2.38	5.34	2.63	2.45	1.18	2.47	1.06
4	6.42	1.08	9.99	2.33	13.7	2.25	6.35	1.30	6.29	1.39
5	7.75	1.04	13.8	2.19	11.4	2.07	7.64	1.14	7.59	1.14
6	7.84	1.04	11.5	2.06	12.5	1.91	7.90	1.09	7.92	1.46
7	7.96	1.07	11.1	1.94	11.2	1.95	8.00	1.17	7.95	1.17
8	8.16	1.09	10.6	1.72	9.96	1.60	7.94	1.14	7.93	1.28
9	8.01	1.06	9.99	1.57	9.10	1.30	8.00	1.12	7.70	1.28
A	6.04	1.06	9.02	2.16	9.09	2.10	6.01	1.13	5.96	1.21

A comparison of the partitioners in terms of the percentage of cut edges leads to similar conclusions (cf. Table 4.5). PMeTiS, Jostle-MS, and Jostle-MD are comparable, but UAMeTiS and DAMeTiS are almost twice as bad. The number of cut edges always increases after a repartitioning since the load imbalance factor has to be reduced.

Our overall conclusions from the results presented in Tables 4.3–4.5 are as follows. PMeTiS is the best partitioner for Sequence_1 since it is very fast and gives the highest quality. UAMeTiS and DAMeTiS are faster partitioners but suffer from poor load balancing quality. Jostle-MS and Jostle-MD, on the other hand, produce high quality subdomains but require a relatively long time to perform the partitioning. In general, we expect global methods to produce higher quality partitions than diffusive schemes, since they have more flexibility in choosing subdomain boundaries.

The remapping times for all five partitioners are presented in Table 4.6. Two remapping strategies are used, resulting in different remapping times at each level. The first strategy uses the default processor mapping given by the respective partitioners, while the second performs processor reassignment based on our heuristic

Table 4.5. Percentage of cut edges before and after mesh partitioning for P=64 using a variety of partitioners for Sequence_1

	PMeTiS		UAMeTiS		DAMeTiS		Jostle-MS		Jostle-MD	
L	Bef	Aft	Bef	Aft	Bef	Aft	Bef	Aft	Bef	Aft
1	6.61	8.95	6.61	17.8	6.61	15.8	6.61	9.04	6.61	9.04
2	10.6	13.2	22.0	25.0	19.4	23.6	10.9	14.4	10.8	13.8
3	13.1	17.1	26.2	29.6	25.0	28.6	14.6	17.0	13.4	19.8
4	9.80	16.4	20.7	31.9	20.3	32.3	9.54	15.1	11.5	15.0
5	10.8	16.0	23.6	30.9	20.6	31.6	9.82	17.4	9.62	15.6
6	9.65	16.7	25.6	30.8	27.2	31.2	10.8	17.3	9.11	15.8
7	9.38	15.8	22.9	31.9	27.9	30.7	10.6	17.8	9.88	17.2
8	9.62	16.0	25.1	32.1	27.2	30.6	10.8	16.9	9.83	14.6
9	9.27	15.8	27.4	31.8	24.4	26.2	10.0	16.3	9.22	14.8
A	9.86	15.1	22.2	29.1	22.1	27.8	10.4	15.7	9.99	15.1

solution of the similarity matrix. It is important to note here that our heuristic strategy uses the w_{remap} weights of the dual graph vertices to minimize the data remapping cost while the partitioners use the w_{comp} weights. Even though the w_{remap} values are the correct ones to use, it is not possible for the current versions of the various partitioners to use them. Several observations can be made from the results. The default remapping times are the fastest for Jostle-MD. PMeTiS is about 17% while UAMeTiS and DAMeTiS are about 25% slower. However, the heuristic remapping times for PMeTiS, Jostle-MS, and Jostle-MD are comparable while those for UAMeTiS and DAMeTiS are about 40% longer. Also note that our heuristic remapper reduces the remapping time by more than 28% for PMeTiS and by about 17% for the Jostle partitioners. However, the improvement is less than 6% for UAMeTiS and about 11% for DAMeTiS.

It is interesting to note that for Sequence_1, a global partitioner like PMeTiS results in a significantly lower remapping overhead than its diffusive counterparts. This seems rather unexpected since the general purpose of diffusive schemes is to minimize the remapping cost. We believe that this discrepancy is due to the high

Table 4.6. Remapping time on an SP2 for P=64 using the default and our heuristic strategies for Sequence_1

	PMeTiS		UAMeTiS		DAMeTiS		Jostle-MS		Jostle-MD	
L	Def	Heu	Def	Heu	Def	Heu	Def	Heu	Def	Heu
1	1.17	1.06	1.25	1.14	1.23	1.12	1.16	1.05	1.16	1.06
2	2.37	1.98	2.34	2.16	2.37	2.02	2.32	1.96	2.32	1.95
3	6.38	4.85	5.73	5.46	5.63	5.24	5.14	4.88	5.07	4.84
4	7.52	6.18	10.9	10.3	13.6	12.4	7.16	6.11	7.24	6.52
5	11.9	7.40	13.4	12.7	12.5	11.2	11.6	7.60	8.28	7.40
6	11.5	7.66	11.8	11.6	13.0	11.9	9.45	7.49	9.16	7.73
7	10.4	8.37	12.7	11.2	11.4	10.6	10.4	7.75	10.6	7.74
8	11.0	7.87	11.1	10.5	10.2	9.83	8.49	7.61	10.1	7.91
9	11.6	7.66	9.83	9.58	9.10	8.88	9.32	7.80	9.24	8.45
A	8.19	5.89	8.77	8.29	8.79	8.13	7.23	5.81	7.02	5.96

growth rate and speed with which our test meshes are evolving. For this class of problems, globally repartitioning the graph from scratch seems to be more efficient then attempting to diffuse the rapidly moving adapted region.

4.2.2 SP2 vs. Origin2000

We next compare the relative performance of the SP2 and the Origin2000. Since we had access to only 32 processors of the Origin2000, experiments on the SP2 were also run using $P = 32$ for this case. We paired the number of partitioners down to two: PMeTiS and DAMeTiS. PMeTiS was chosen because it was the best partitioner overall. DAMeTiS was chosen over the Jostle partitioners since faster repartitioning is more important than higher quality in an adaptive-grid scenario. The partitioning and the remapping times using our heuristic remapping strategy for Sequence_1 are presented in Table 4.7. Consistent with the results in Table 4.3, DAMeTiS is slightly faster than PMeTiS on both machines. Consistent with the results in Table 4.1, run times on the Origin2000 are about half the corresponding times on the SP2. The DAMeTiS remapping times are higher than PMeTiS, but not as bad as in Table 4.6. Finally, the remapping times are about three times faster on the Origin2000 than on the SP2 as was also shown earlier in Table 4.2.

Table 4.7. Partitioning and remapping times on the SP2 and the Origin2000 for P=32 using PMeTiS and DAMeTiS for Sequence_1

	Partitioning				Heuristic Remapping			
	PMeTiS		DAMeTiS		PMeTiS		DAMeTiS	
L	SP2	O2000	SP2	O2000	SP2	O2000	SP2	O2000
1	0.35	0.45	0.36	0.44	1.43	0.47	1.58	0.50
2	0.42	0.20	0.48	0.23	3.19	1.10	2.87	1.05
3	0.68	0.33	0.68	0.30	5.49	1.82	8.86	2.68
4	0.96	0.47	0.90	0.44	11.0	3.66	17.5	6.57
5	0.75	0.41	1.00	0.40	14.1	4.62	17.7	6.30
6	1.09	0.50	0.75	0.43	15.4	4.78	14.9	5.83
7	0.79	0.42	0.75	0.34	15.4	4.78	15.3	5.04
8	1.12	0.37	0.80	0.32	15.0	4.93	13.3	4.65
9	0.86	0.34	0.80	0.34	15.7	5.04	14.9	4.03
A	0.78	0.39	0.72	0.36	10.7	3.47	11.9	4.07

The quality of load balancing for this experimental case is presented in Table 4.8. Theoretically, these results should be identical on both machines. However, since PMeTiS and DAMeTiS use pseudo-random numbers in their codes, the results were not uniform due to different seeds on the SP2 and the Origin2000. The results shown in Table 4.8 are obtained on the Origin2000. PMeTiS is once again better than DAMeTiS, both in terms of the load imbalance factor and the percentage of cut edges. These results are consistent with those shown in Tables 4.4 and 4.5; however, the values are smaller here. The load imbalance factors are lower because fewer processors are used. The percentages of cut edges are smaller since the surface-to-volume ratio decreases with the number of partitions.

4.2.3 Coarse vs. fine initial mesh

Figure 4.7 presents the partitioning and remapping times using PMeTiS for the two mesh granularities, Sequence_1 and Sequence_2. Remapping results are presented only for our heuristic remapping strategy. A couple of observations can be

Table 4.8. Load imbalance factor and percentage of cut edges before and after mesh partitioning for P=32 using PMeTiS and DAMeTiS for Sequence_1

	Load imbalance factor				Percentage of cut edges			
	PMeTiS		DAMeTiS		PMeTiS		DAMeTiS	
L	Bef	Aft	Bef	Aft	Bef	Aft	Bef	Aft
1	3.58	1.01	3.58	1.88	4.65	6.28	4.65	15.7
2	2.17	1.04	3.95	2.12	7.66	9.65	19.3	20.5
3	2.41	1.06	4.90	2.12	9.57	13.2	21.1	25.3
4	6.14	1.05	9.82	1.87	7.99	12.2	17.1	28.2
5	7.31	1.03	10.2	1.68	6.76	11.8	29.1	26.5
6	7.88	1.05	9.12	1.41	7.15	11.1	25.3	24.4
7	7.86	1.04	7.82	1.11	6.47	11.3	20.6	14.2
8	8.02	1.04	6.66	1.05	6.50	11.5	10.0	13.9
9	7.92	1.05	6.61	1.05	6.21	10.9	9.41	14.2
A	5.92	1.04	6.96	1.59	7.00	10.9	17.4	20.3

made from the resulting graphs. First, when comparing the two sequences, results show that the finer mesh increases both the partitioning and the remapping times by almost an order of magnitude. This is expected since the initial fine mesh is eight times the size of the initial coarse mesh. The larger graph is thus more expensive to partition and requires more data movement during remapping. Second, increasing the number of processors from 16 to 64 does not have a major effect on the partitioning times, but causes an almost three-fold reduction in the remapping times. This indicates that our load balancing strategy will remain viable on a large number of processors.

Figure 4.8 presents the quality of load balancing for Sequence_1 and Sequence_2 using PMeTiS. Load balancing quality is again measured in terms of the load imbalance factor and the percentage of cut edges. For all the cases, the partitioner does an excellent job of reducing the imbalance factor to near unity. Using a finer mesh has a negligible effect on the imbalance factor after load balancing, but requires a substantially longer repartitioning time (cf. Fig. 4.7). The percentage of cut edges always increases with the number of processors. This is expected since the

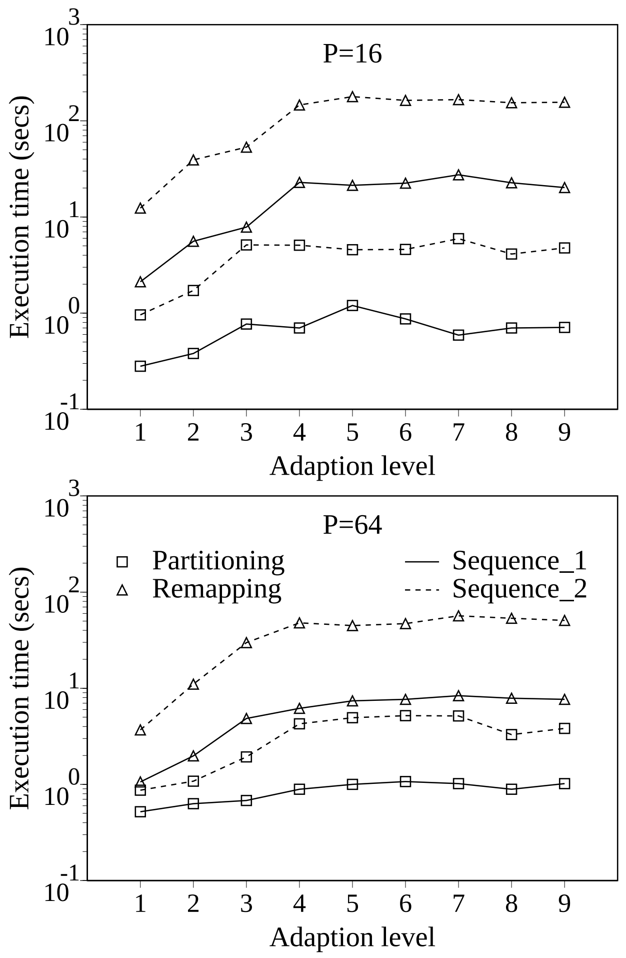


Figure 4.7. PMeTiS partitioning and remapping times using the heuristic strategy for P=16 and 64 on an SP2 for Sequence_1 and Sequence_2.

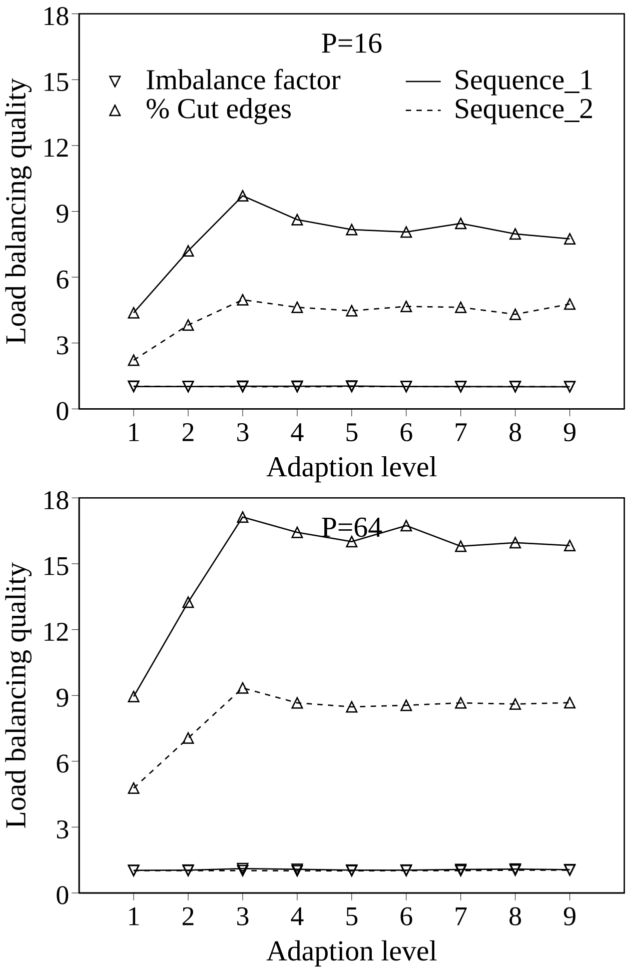


Figure 4.8. Load imbalance factor and percentage of cut edges after mesh partitioning using PMeTiS for P=16 and 64 for Sequence_1 and Sequence_2. Note that the imbalance factor curves for the two sequences are overlaid.

surface-to-volume ratio increases with the number of partitions. Also notice that the percentage of cut edges generally grows with each level of adaption, and then stabilizes when the mesh size stabilizes. This is because successive adaptions create a complex distribution of computationally-heavy nodes in the dual graph, thereby requiring partitions to have more complicated boundaries to achieve load balance. This increases the surface-to-volume ratio of the partitions, resulting in a higher percentage of cut edges. The finer mesh consistently has a smaller percentage of cut edges because the partitioner has a wider choice of edges to find a better cut. However, we believe that this savings in the number of cut edges does not warrant the significantly higher overhead of the finer mesh. Note that a more precise flow solution can be achieved using the fine mesh since it was adapted one level deeper than the coarse grid. Nonetheless, we expect our overall conclusions to remain the same, even if an additional adaption was performed on the coarse mesh.

4.2.4 Growing vs. stable mesh

Lastly, we compare the performance of PMeTiS and DAMeTiS for Sequence_3 on 32 processors of the SP2. The reason for this experiment was to investigate the effect of our load balancing strategy on a mesh that continuously grows in size through the course of adaption. The partitioning and the remapping times are presented in Table 4.9. A comparison with the results in Table 4.7 shows that the partitioning times for both partitioners are almost unchanged. This is because both Sequence_1 and Sequence_3 use the same initial mesh; thus, the partitioners work on dual graphs that are topologically identical. The remapping times, however, are significantly higher for Sequence_3 because of a much larger adapted mesh. Even though the adaption region is moving with a lower velocity here than for Sequence_1, the mesh is growing very rapidly, gaining more than two orders of magnitude in only nine adaption levels. Our heuristic remapper reduces the remapping time by more than 23% for PMeTiS and by almost 17% for DAMeTiS. Once again, the global repartitioning strategy using PMeTiS produces a lower remapping overhead than the diffusive scheme.

Table 4.9. Partitioning and remapping times on an SP2 for P=32 using PMeTiS and DAMeTiS for Sequence_3

	Partitioning		Remapping			
			PMeTiS		DAMeTiS	
L	PMeTiS	DAMeTiS	Def	Heu	Def	Heu
1	0.34	0.59	1.30	1.26	1.15	1.18
2	0.32	0.34	1.45	1.27	1.53	1.38
3	0.34	0.38	2.17	1.72	2.39	1.95
4	0.60	0.46	5.68	4.52	4.80	4.47
5	0.88	0.75	15.1	10.6	17.1	14.3
6	1.35	0.72	23.9	16.4	32.4	27.3
7	1.25	1.32	44.2	29.4	58.6	40.6
8	1.18	0.93	53.8	39.3	86.9	71.2
9	0.95	0.76	50.5	47.8	81.7	75.4
A	0.80	0.69	22.0	16.9	31.8	26.4

The quality of load balancing is presented in Table 4.10. PMeTiS is once again significantly better than DAMeTiS in terms of the load imbalance factor. Compared to the corresponding results in Table 4.8, the imbalance factor after mesh repartitioning is higher, particularly for DAMeTiS. This is due to the lower speed of the adapted region, which increases the maximum values of w_{comp} and w_{comm} in the dual graph. This, in turn, limits the efficacy of the partitioner to balance the mesh, since certain nodes have become very heavy. An additional side effect is that the percentage of cut edges are significantly worse for Sequence_3 than for the higher speed simulation of Sequence_1, shown in Table 4.8. Nonetheless, a near perfect load balance is achieved by PMeTiS for this test case, even though it is partitioning the dual of an initial mesh which has grown by over 120-fold in only nine adaptions. This indicates that our dual graph scheme with adjustable vertex and edge weights can be successfully used even when the mesh is growing significantly and rapidly.

Table 4.10. Load imbalance factor and percentage of cut edges before and after mesh partitioning for P=32 using PMeTiS and DAMeTiS for Sequence_3

	Load imbalance factor				Percentage of cut edges			
	PMeTiS		DAMeTiS		PMeTiS		DAMeTiS	
L	Bef	Aft	Bef	Aft	Bef	Aft	Bef	Aft
1	1.89	1.03	1.89	1.13	4.70	4.73	4.70	6.75
2	4.46	1.03	4.31	1.39	4.75	6.85	8.82	15.5
3	3.26	1.04	3.78	2.37	11.6	20.8	29.5	25.8
4	2.17	1.08	3.99	2.75	28.6	34.4	36.6	33.3
5	2.31	1.03	4.33	3.08	34.2	47.7	33.6	42.2
6	3.80	1.08	5.69	2.59	40.4	49.6	41.7	44.8
7	3.59	1.15	3.72	2.97	41.3	48.9	39.4	44.4
8	4.06	1.13	8.26	2.42	37.6	44.4	42.4	42.6
9	4.45	1.15	5.26	2.09	37.2	45.5	36.8	44.4
A	3.33	1.08	4.58	2.31	26.7	33.7	30.4	33.3

CHAPTER 5

SUMMARY AND FUTURE WORK

5.1 Summary

Simulation of large-scale transient flows around complex geometric bodies is a common challenge in many fields of computational fluid dynamics. To address these problems there is a demonstrable need for unstructured mesh adaptivity on multiprocessor systems. Efficient implementations of these procedures is a complex task primarily due to the load imbalance resulting from the dynamically changing nonuniform grids. In this thesis we have developed PLUM, an automatic portable framework for performing large-scale numerical computations in a message-passing environment.

The most significant contribution of this thesis is the development and validation of a load balancing methodology with a global view. In Chapter 1, we presented a historical overview of techniques used to balance adaptive unstructured mesh computations. Most previous efforts have relied on locally diffusive schemes, since it was generally considered too expensive to repartition the entire domain in the inner loop of an adaptive flow calculation. We also assert that local iterative techniques are not ideally suited for dynamically balancing unsteady flows. These applications are prone to dramatically shifting the load distribution between adaption phases, causing small regions of the domain to suddenly incur high computational costs. Local diffusion techniques are therefore required to perform many iterations before global convergence, or accept an unbalanced load in exchange for faster performance. Also, by limiting task movement to nearest neighbors, a finite element may have to make several hops before arriving at its final destination. Finally, global schemes will generally produce superior subdomain quality, since they are not restricted to nearest neighbor communications. In order to develop an effective global balancing scheme, we needed to mitigate the potentially high cost of partitioning and data remapping. Additionally, a successful framework has be portable and remain viable on a large number of processors. We have demonstrated that PLUM achieves these criteria on realistic-sized meshes for both steady and unsteady simulations.

In Chapter 2, we presented our distributed memory implementation of the tetrahedral mesh adaption scheme developed by Biswas and Strawn [12]. The parallel code consists of approximately 3,000 lines of C++ with MPI which wraps around the original version written in C. The serial code was left almost completely unchanged except for a few lines which interface with the parallel wrapper. This allowed us to design the parallel version using the serial code as a building block. The object-oriented approach maintains to build a clean interface between the two layers of the program while maintaining efficiency. Only a slight increase in memory was necessary to keep track of the global mappings and shared processor lists for objects located on partition boundaries.

Six refinement and two coarsening cases were presented with varying fractions of a realistic-sized domain being targeted for refinement. We have shown extremely promising parallel performance of more than 52.5X on 64 processors of an SP2 when about 60% of the computational mesh used to simulate a helicopter acoustics experiment was dynamically refined, using a solution-based error indicator. Performance was significantly improved by repartitioning and remapping the mesh in a load-balanced fashion after edges were targeted for refinement but before performing the actual subdivision.

Chapter 3 presented PLUM, our dynamic load balancing framework. Several salient features of this methodology were described: (i) a dual graph representation, (ii) parallel mesh repartitioning, (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. Large-scale scientific computations on an SP2 showed that load balancing can significantly reduce flow solver times over non-balanced loads. With multiple mesh adaptions, the gains realized with load balancing may be even more dramatic.

Using the dual graph representation of the initial mesh for the purpose of partitioning is one of the key features of this work. 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 complexity of the dual graph remains unchanged during the course of an adaptive computation. Therefore, the repartitioning times depend only on the initial problem size and the number of partitions - but not on the size of the adapted mesh.

We performed two different tests on PLUM using a realistic-sized computational mesh on an SP2. 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 indicated that by using a high quality parallel partitioner to rebalance the mesh, a perfectly load balanced flow solver is guaranteed with minimal communication overhead.

An important contribution of this research is our development of the processor reassignment phase. The goal is to find a mapping between partitions and processors such that the data 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. Various cost functions are usually needed to solve the processor reassignment problem for different architectures. We present three general metrics:

`TotalV`, `MaxV`, and `MaxSR` which model the remapping cost on most multiprocessor systems. The metric `TotalV` assumes that by reducing network contention and the total number of elements moved, the remapping time will be reduced. The `MaxV` and `MaxSR` metrics, 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. In general, the overall objective function may need to use a combination of metrics to effectively incorporate all related costs. Optimal solutions for all three metrics, as well as a heuristic approach were implemented. It was shown that our heuristic algorithm quickly finds high quality solutions for all our metrics. Additionally, strong theoretical bounds on the heuristic time and solution quality were presented.

Once the reassignment problem is solved, a model is needed to quickly predict the expected redistribution cost on a given architecture, to ensure that it is more than compensated for by the computational gain of balanced partitions. 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. 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. We developed a new remapping cost model for the SP2, and quantitatively validated its accuracy in predicting redistribution overhead. Results indicated that reducing the bottleneck, rather than the aggregate, overhead guarantees a reduction in the total redistribution time.

The remapping phase is responsible for physically moving data when it is reassigned to a different processor, and is generally the most expensive phase of any load balancing strategy. In this thesis, we made the key observation 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 significant cost savings can be achieved. This efficient remapping strategy resulted in almost a four-fold cost savings for data movement when 60% of the computational mesh was refined.

Several experiments were performed in Chapter 4 to verify the effectiveness of PLUM on sequences of dynamically adapted unstructured grids. Results demonstrated that our framework works well for both steady and unsteady adaptive problems with many levels of adaption, even when using a coarse initial mesh. We showed that our dual graph scheme with adjustable vertex and edge weights can be successfully used even when the mesh is growing significantly and rapidly. A comparison of coarse and fine initial grids was presented to evaluate the relationship between dual mesh granularity and load balancing performance. We found that a finer starting mesh may be used to achieve lower edge cuts and marginally better load balance, but is generally not worth the increased partitioning and data remapping times.

Portability was examined by comparing results between the distributed-memory system of the IBM SP2, and the Scalable Shared-memory MultiProcessing (S2MP) architecture of the SGI/Cray Origin2000. The refinement procedure showed promising parallel results and achieved an efficiency of more than 87% on 32 processors of both the SP2 and the Origin2000, for our largest test case. Additionally, the performance of all our load balancing modules were similar on both architectures. These results demonstrated that PLUM can be effectively ported to different platforms without the need for any code modifications. We also applied the SP2 redistribution cost model to the Origin2000, but with limited success. Future research will address the development of a more comprehensive remapping cost model for the Origin2000.

Finally, we conducted a repartitioning analysis by examining the performance of five, state-of-the-art parallel partitioners within PLUM. We found that a global partitioner like PMeTiS significantly outperforms its diffusive counterparts, for both remapping overhead and subdomain quality. In general, global methods are expected to produce higher quality partitions than diffusive schemes, since they have more flexibility in choosing subdomain boundaries. We believe that the discrepancy in remapping overhead is due to the high growth rate and speed with which our test meshes evolved. These results validate our earlier claim that for this class of unsteady problems, globally repartitioning the graph from scratch is more efficient then attempting to diffuse the rapidly moving adapted region. Additionally, we showed that the data redistribution overhead can be reduced by applying our heuristic processor reassignment algorithm to the default partition-to-processor mapping given by all five partitioners.

5.2 Future Work

There are many extensions that can be made to the work presented here. First, we plan to interface PLUM with a parallel flow solver system. The combination of these two components should allow us to compute solutions for systems which were previously unsolvable. Additionally, new insight will be gained by observing the sustained performance of PLUM. We also plan to investigate the relationship between subdomain quality and flow solver performance. Currently the total edge cut is used as the standard metric for evaluating partition quality. We believe that a more sophisticated model is needed in order to accurately predict flow solver overhead.

Several extension can be made to the processor reassignment phase. In Sec. 3.5 we developed a technique for assigning $F \geq 1$ unique partitions to each processor using the `TotalV` metric. A similar algorithm for the `MaxV` and `MaxSR` metrics could be developed, since it is currently limited a to one-to-one mapping between partitions and processors. An extensive analysis could determine the effectiveness of setting $F > 1$. By having multiple partitions assigned to each processor we may reduce the remapping overhead, at the expense of higher partitioning times and disjoint subdomains. We can also extend the similarity matrix construction and processor reassignment phase to consider processor locality. Some architectures, such as the hypercube or 3D-torus, can require multiple message hops between two communicating processors. Additionally, hierarchical interconnection layers can affect the relative cost of each hop. These additional parameters could be incorporated into our framework for these architectures, in order to minimize and predict remapping overhead.

Finally, we would like to compare our message-passing implementation of PLUM with other programming paradigms, such as CC-NUMA and multithreading. A drawback of our MPI load balancing system is the high computation and communication overhead incurred during redistribution. A multithreading approach may be used as a means of exploring concurrency in the processor level in order to tolerate synchronization costs inherent in traditional nonthreaded systems. Preliminary results indicate that multithreading can be used as a mechanism to mask the overheads required for the dynamic balancing of processor workloads, with the computations required for the actual numerical solution of PDEs [17]. Unfortunately multithreading complicates program complexity and makes code reusability a difficult task. Another drawback of the current PLUM implementation is the code complexity resulting from explicit message passing. CC-NUMA offers the advantage of a global address space with automatic page migration. As a result, code development time should be considerably lower than the MPI implementation. A potential disadvantage of this approach, however, is the degradation of parallel performance as the number of processors increases. A comparison of all three programming methodologies would provide an extremely valuable analysis.

BIBLIOGRAPHY

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

[2] K. Baumgartner, J. Stankovic, and K. Zhao, Distributed scheduling of tasks with deadlines and resource requirements. *IEEE Trans. Comput.,* pp. 1110-1123, 1989.

[3] M.J. Berger, and J.S. Saltzman, AMR on the CM-2. *Appl. Numer. Math.,* 14:1–3, pp. 239–253, April 1994.

[4] K.S. Bey, J.T. Oden, and A. Patra, A parallel hp-adaptive discontinuous Galerkin method for hyperbolic conservation laws. *Appl. Numer. Math.,* 20:4, pp. 321–336, April 1996.

[5] K.V.S. Bhat, An $O(n^{2.5}\log_2 n)$ time algorithm for the bottleneck assignment problems, *unpublished report*, AT&T Bell Laboratories, Napierville, IL, 1984.

[6] E. Biagioni, Scan directed load balancing. *PhD thesis,* Department of Computer Science, University of North Carolina at Chapel Hill, 1991.

[7] R. Biswas, Parallel and adaptive methods for hyperbolic partial differential systems. *Ph.D. thesis,* Department of Computer Science, Rensselaer Polytechnic Institute, 1991.

[8] R. Biswas and L. Dagum, Parallel implementation of an adaptive scheme for 3d unstructured grids on a shared-memory multiprocessor. *Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers,* Elsevier Science, Amsterdam, The Netherlands, pp. 489–496, 1996.

[9] R. Biswas, K.D. Devine, and J.E. Flaherty, Parallel, adaptive finite element methods for conservation laws. *Appl. Numer. Math.,* 14:1–3, pp. 255–283, April 1994.

[10] R. Biswas, L. Oliker, and A. Sohn, Global load balancing with parallel mesh adaption on distributed-memory systems. *Proceedings of Supercomputing '96,* Pittsburgh, Pennsylvania, Nov. 17-22, 1996.

[11] R. Biswas and L. Oliker, Load balancing sequences of unstructured adaptive grids. *4th International Conference on High Performance Computing,* 1997, to appear.

[12] R. Biswas and R.C. Strawn, A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics,* 13, pp. 437–452, 1994.

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

[14] Y. Boglaev, Exact dynamic load balancing of MIMD architectures with linear programming algorithms. *Parallel Computing,* Vol. 18, No. 6, pp. 615-623, 1991.

[15] C. Bottasso, H. Cougny, M. Dindar, J. Flaherty, C. Ozturan, Z. Rusak, and M. Shephard, Compressible aerodynamics using a parallel adaptive time-discontinuous galerkin least-squares finite element method. *12th AIAA Applied Aerodynamics Conference,* Paper 94-1888, 1994.

[16] J.G. Castanos and J.E. Savage, The dynamic adaptation of parallel mesh-based computation. *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing,* SIAM Activity Group on Supercomputing, Minneapolis, MN, 1997.

[17] N. Chrisochoides, Multithreaded model for the dynamic load-balancing of parallel adaptive PDE computations. *Applied Numerical Mathematics*, 20, pp. 321–336, 1996.

[18] 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, pp. 157–182, 1994.

[19] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, 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.

[20] G. Cybenko, Dynamic load balancing for distributed-memory multiprocessors. *Journal of Parallel and Distributed Computing,* 7, pp. 279–301, 1989.

[21] S.K. Das, D.J. Harvey, and R. Biswas, Adaptive load-balancing algorithms using symmetric broadcast networks: performance study on an IBM SP2. *26th International Conference on Parallel Processing,* pp. 360–367, 1997.

[22] Y. Deng, R. McCoy, R. Marr, and R. Peierls, An unconventional method for load balancing. *7th SIAM Conference on Parallel Processing for Scientific Computing,* pp. 605–610, 1995.

[23] K. Devine and J. Flaherty, Parallel adaptive hp-refinement techniques for conservation laws. *Applied Numberical Mathematics,* Vol. 20, No. 4, pp. 367-387, 1996.

[24] K. Devine, J. Flaherty, S. Wheat, and A. Maccabe, A massively parallel adaptive finite element method with dynamic load balancing. *Supercomputing,* pp. 2–11, 1993.

[25] W. Donath and A. Hoffman, Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin,* Vol. 15, pp. 938-944, 1972.

[26] M. Fiedler, A property of eigenvectors of nonnegative symmetric matrices and its applications to graph theory. *Czech Math Journal,* Vol. 25, pp. 619-633, 1975.

[27] M.L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in network optimization. *J. Assoc. Comput.,* Mach. 34 , pp. 596–615, 1987.

[28] H.N. Gabow and R.E. Tarjan, Faster scaling algorithms for network problems, *Technical Report CS-TR-111-87,* Department of Computer Science, Princeton University, Princeton, NJ 1987.

[29] H.N. Gabow and R.E. Tarjan, Algorithms for two bottleneck optimization problems. *J. Algorithms,* 9, pp. 411–417, 1988.

[30] M. Garey and D. Johnson, Computers and intractability: a guide to the theory of np-completeness. *Freeman Press,* San Francisco, 1979.

[31] B. Ghosh and S. Muthukrishnan, Dynamic load balancing in parallel and distributed networks by random matchings. *6th ACM Symposium on Parallel Algorithms and Architectures,* pp. 226–235, 1994.

[32] S. Hammond, Mapping unstructured grid computations to massively parallel computers. *PhD thesis,* Rensselaer Polytechnic Institute, Troy, New York, 1992.

[33] 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, pp. 303-312, 1995.

[34] B. Hendrickson and R. Leland, The Chaco user's guide — Version 2.0. *Sandia National Laboratories Technical Report,* SAND94-2692, 1994.

[35] B. Hendrickson and R. Leland, Multidimensional spectral load balancing. *Sandia National Laboratories Technical Report,* SAND93-0074, 1993.

[36] B. Hendrickson and R. Leland, A multilevel algorithm for partitioning graphs. *Sandia National Laboratories Technical Report,* SAND93-1301, 1993.

[37] G. Horton, A multi-level diffusion method for dynamic load balancing. *Parallel Computing,* 19, pp. 209–229, 1993.

[38] M.T. Jones and P.E. Plassmann, Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. *Proc. Scalable High Performance Computing Conference,* IEEE Computer Society, Knoxville, TN, pp. 478–485, 1994.

[39] Y. Kallinderis, and A. Vidwans, Generic parallel adaptive-grid Navier-Stokes algorithm. *AIAA J.,* 32:1, pp. 54–61, January 1994.

[40] G. Karypis, and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs. *Department of Computer Science, Technical Report,* 95-035, University of Minnesota, Minneapolis, MN, 1995.

[41] G. Karypis and V. Kumar, Parallel multilevel k-way partitioning scheme for irregular graphs. *Department of Computer Science, Technical Report,* 96-036, University of Minnesota, Minneapolis, MN, 1996.

[42] G.A. Kohring, Dynamic load balancing for parallelized particle simulations on MIMD computers. *Parallel Computing,* 21, pp. 683–693, 1995.

[43] B. Kernighan and S. Lin, An effective heuristic procedure for partitioning graphs. *The Bell System Technical Journal,* pp. 291-308, 1970.

[44] J. Keyser and D. Roose, Grid partitioning by inertial recursive bisection. *Report TW 174, K. U. Leuven,* Department of Computer Science, Belgium, July 1992.

[45] S. Khuri and A. Baterekh, Genetic algorithms and discrete optimization. *Methods of Operations Research,* Vol. 64, pp. 133-142, 1991.

[46] S. Kirkpatrick, C. Gelatt and M. Vecchi, Optimization by simulated anealing. *Science,* pp. 671-680, 1983.

[47] E. Leiss and H. Reddy, Distributed load balancing: design and performance analysis. *W. M. Keck Research Computation Laboratory,* Vol. 5, pp. 205-270, 1989.

[48] R. Lohner, An adaptive finite element scheme for transient problems in CFD. *Comp. Mech. Eng.,* Vol. 61., pp. 323-338, 1987.

[49] N. Mahapatra and S. Dutt, Random Seeking: a general, efficient, and informed randomized scheme for dynamic load balancing. *International Parallel Processing Symposium,* pp. 881-885, 1996.

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

[51] C. Ou, S. Ranka, and G. Fox, Fast mapping and remapping algorithm for irregular and adaptive problems. *Proceedings of the 1993 International Conference on Parallel and Distributed Systems,* 1993.

[52] L. Oliker and R. Biswas, Efficient load balancing and data remapping for adaptive grid calculations. *9th ACM Symposium on Parallel Algorithms and Architectures,* pp. 33–42, 1997.

[53] 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,* Springer-Verlag, LNCS 1117, pp. 35–47, 1996.

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

[55] J.J. Quirk, A parallel adaptive grid algorithm for computational shock hydrodynamics. *Appl. Numer. Math.,* 20:4, pp. 427–453, April 1996.

[56] P.M. Selwood, N.A. Verhoeven, J.M. Nash, M. Berzins, N.P. Weatherill, P.M. Dew, and K. Morgan, Parallel mesh generation and adaptivity: partitioning and analysis. *Parallel Computational Fluid Dynamics: Algorithms and Results Using Advanced Computers,* Elsevier Science, Amsterdam, The Netherlands, pp. 166–173, 1997.

[57] M. Sharp and C. Farhat, TOP/DOMDEC, a totally object oriented program for visualisation, domain decomosition and parallel processing. *User's Manual,* PGSoft and The University of Colorado, Boulder, Feb, 1994.

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

[59] A. Sohn, R. Biswas and H. Simon, A dynamic load balancing framework for unstructured adaptive computations on distributed-memory multiprocessors. *8th ACM Symposium on Parallel Algorithms and Architectures,* 1996, to appear.

[60] H.D. Simon, A. Sohn, and R. Biswas, HARP: A fast spectral partitioner. *Proc. 9th ACM Symposium on Parallel Algorithms and Architectures.* ACM SIGACT and SIGARCH, Newport, RI, pp. 43-52, 1997.

[61] H. Simon, Partitioning of unstructured problems for parallel processing. *Computer Systems Engineering,* Vol. 2, pp. 135-148, 1991.

[62] K. Schloegel, G. Karypis, and V. Kumar, Multilevel diffusion schemes for repartitioning of adaptive meshes. *Department of Computer Science, Technical Report,* 97-013, University of Minnesota, Minneapolis, MN, 1997.

[63] A. Sohn, R. Biswas, and H.D. Simon, Impact of load balancing on unstructured adaptive grid computations for distributed-memory multiprocessors. *8th IEEE Symposium on Parallel and Distributed Processing,* pp. 26–33, 1996.

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

[65] R.C. Strawn, R. Biswas, and M. Garceau, Unstructured adaptive mesh computations of rotorcraft high-speed impulsive noise. *Journal of Aircraft,* 32, pp. 754–760, 1995.

[66] L.G. Valiant, A bridging model for parallel computation. *Communications of the ACM,* 33, pp. 103–111, 1990.

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

[68] Van R. Driessche and D. Roose, An improved spectral bisection algorithm and its application to dynamic load balancing. *Parallel Computing,* Vol. 21, No. 1, pp. 29-48, 1994.

[69] D. Vanderstraeten, C. Farhat, P. Chen, R. Keunings, and O. Zone, A retrofit based methodology for the fast generation and optimization of large-scale mesh partitions: beyons the minimum interface size criterion. *University of Colorado, Technical Report,* CU-CAS-94-18, 1994.

[70] A. Vidwans, Y. Kallinderis, and V. Venkatakrishnan, Parallel dynamic load-balancing algorithm for three-dimensional adaptive unstructured grids. *AIAA Journal,* 32, pp. 497–505, 1994.

[71] C. Walshaw, M. Cross, and M.G. Everett, A localised algorithm for optimizing unstructured mesh partitions. *Int. J. Supercomputer Appl.,* 9(4), pp. 280-295, 1995.

[72] C. Walshaw, M. Cross, and M.G. Everett, Parallel dynamic graph-partitioning for unstructured meshes. *School of Computing and Mathematical Sciences, Technical Report,* 97/1M/20, University of Greenwich, London, UK, 1997.

[73] J. Watts, M. Rieffel and S. Taylor, Practical dynamic load balancing for irregular problems. *Parallel Algorithms for Irregularly Structured Problems,* Springer-Verlag, LNCS 1117, 1996.

[74] M. Willebeek-LeMair and A. Reeves, Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems,* Vol. 4, No. 9, 1993.