EXPERIMENTS WITH REPARTITIONING AND LOAD BALANCING ADAPTIVE MESHES

RUPAK BISWAS* AND LEONID OLIKER†

Abstract. Mesh adaption is a powerful tool for efficient unstructured-grid computations but causes load imbalance on multiprocessor systems. To address this problem, we have developed **PLUM**, an automatic portable framework for performing adaptive large-scale numerical computations in a message-passing environment. This paper 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 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.

Key words. Dynamic load balancing, graph partitioning, unstructured mesh adaption, data remapping, redistribution cost model.

AMS(MOS) subject classifications. 68Q22, 65M50, 90C35.

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. Nontheless, 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 [4,5,6,7,8,9,11,19,20]; however, most of them lack a global view of loads across processors.

Given our goal to build a portable system for efficiently performing large-scale adaptive numerical calculations in a parallel message-passing

[*] MRJ Technology Solutions, NASA Ames Research Center, Moffett Field, CA 94035. The work of the first author was supported by NASA under Contract Number NAS 2-14303.

[†] Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA 94035. The work of the second author was supported by NASA under Contract Number NAS 2-96027.

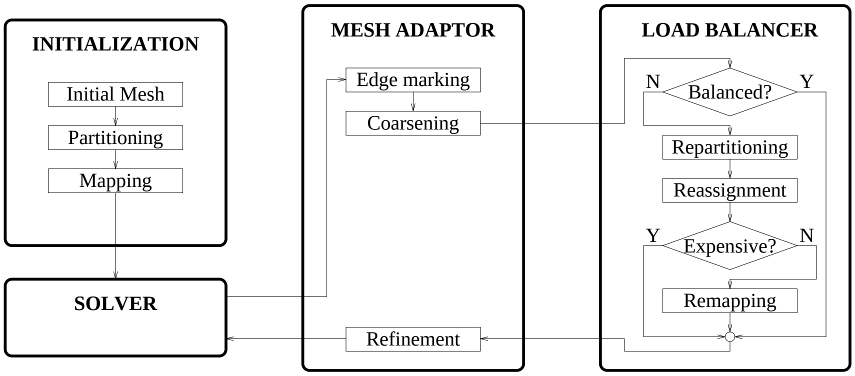


FIG. 1.1. *Overview of* PLUM, *our framework for parallel adaptive numerical computation.*

environment, a novel method has been developed that dynamically balances the processor workloads with a global view. Figure 1.1 depicts our framework, called PLUM, for such an automatic system. The mesh is first partitioned and mapped among the available processors. A numerical solver then runs for several iterations, updating solution variables. Once an acceptable solution is obtained, a mesh adaption procedure is invoked. It first targets edges for coarsening and refinement based on an error indicator computed from the numerical solution. The old mesh is then coarsened, resulting in a smaller grid. Since edges have already been marked for refinement, it is possible to exactly predict the new mesh before actually performing the refinement step. Program control is thus passed to the load balancer at this time. A quick evaluation step determines if the new mesh will be so unbalanced as to warrant a repartitioning. If the current partitions will remain adequately load balanced, control is passed back to the subdivision phase of the mesh adaptor. Otherwise, a repartitioning procedure is used to divide the new mesh into subgrids. The new partitions are then reassigned to the processors in a way that minimizes the cost of data movement. If the remapping cost is compensated by 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 refined and the numerical calculation is restarted.

Extensive details of the parallel mesh adaption scheme, called 3D_TAG, that is used in this work is given in [13]. The parallel version consists of C++ and MPI code wrapped around the original serial mesh adaption program [3]. An object-oriented approach allowed a clean and efficient implementation. 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.

2. Dynamic load balancing. PLUM is a novel method to dynamically balance the processor workloads with a global view. Results reported earlier either focused on fundamental load balancing issues [16] or various refinement strategies [2,12] to demonstrate the viability and effectiveness of our framework. A model that accurately predicts the total cost of data redistribution on an SP2 given the number of tetrahedral elements that have to be moved among processors was presented in [1]. This paper presents the application of PLUM to three sequences of dynamically adapted unstructured grids. Portability is investigated by comparing results on an SP2 and an Origin2000. In addition, the performance of five state-of-the-art partitioning algorithms that can be used within PLUM are examined.

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 significantly reduces the cost of remapping and mesh subdivision; and (v) accurate metrics estimate and compare the computational gain and the redistribution cost of having a balanced workload after each mesh adaption step.

2.1. Dual graph of initial mesh. Using the dual of the initial computational mesh for the purpose of dynamic load balancing is one of the key features of this work. Each dual graph vertex has two weights associated with it. The computational weight, w_{comp}, models the workload for the corresponding element. The remapping weight, w_{remap}, models the cost of moving the element from one processor to another. The weight w_{comp} is set to the number of leaf elements in the refinement tree because only those elements that have no children participate in the numerical computation. The weight w_{remap}, however, is set to the total number of elements in the refinement tree because all descendants of the root element must move with it from one partition to another, if so required. Every edge of the dual graph also has a weight, w_{comm}, that models the runtime interprocessor communication. The value of w_{comm} is set to the number of faces in the computational mesh that corresponds to the dual graph edge. The mesh connectivity, w_{comp}, and w_{comm} together determine how dual graph vertices should be grouped to form partitions that minimize both the disparity in the partition weights and the runtime communication. The w_{remap} determines how partitions should be assigned to processors such that the cost of data redistribution is minimized. New computational grids obtained by adaption are translated to w_{comp} and w_{remap} for every vertex and to w_{comm}

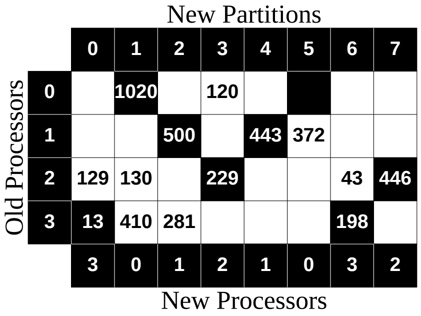
for every edge in the dual mesh.

2.2. Parallel mesh repartitioning. If a preliminary evaluation step determines that the dual graph with a new set of w_{comp} is unbalanced, the mesh needs to be repartitioned. A good partitioner should minimize the total execution time by balancing the computational loads and reducing the interprocessor communication time. In addition, the repartitioning phase must be performed very rapidly for our load balancing framework to be viable. 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.

PMeTiS [10] and Jostle-MS [21] are global partitioners which make no assumptions on how the graph is initially distributed among the processors. Both methods are multilevel k-way partitioning algorithms that reduce the size of the graph by collapsing vertices and edges, partition the smaller problem, and uncoarsen the graph back to the original size. 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 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 [15], DAMeTiS [15], and Jostle-MD [21] are diffusive 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.

2.3. Processor reassignment. Once new partitions are obtained, they must be mapped to processors such that the redistribution cost is minimized. In general, the number of new partitions is an integer multiple F of the number of processors. Each processor is then assigned F unique partitions. The first step toward processor reassignment is to compute a similarity measure S that indicates how the remapping weights w_{remap} of the new partitions are distributed over the processors. It is represented as a matrix where entry S_{ij} is the sum of the w_{remap} of all the dual graph



	New Partitions							
Old Processors	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	
New Processors	3	0	1	2	1	0	3	2

FIG. 2.1. *A similarity matrix after processor reassignment using the heuristic algorithm and the* `TotalV` *metric.*

vertices in new partition j that already reside on processor i. A similarity matrix for $P = 4$ and $F = 2$ is shown in Fig. 2.1. Only the non-zero entries are shown.

The goal of the processor reassignment phase is to find a mapping between partitions and processors such that the data redistribution cost is minimized. Various cost functions are usually needed to solve this problem for different architectures. In [12], we investigated two general metrics: `TotalV`, that minimizes the total volume of data moved among all processors, and `MaxV`, that minimizes the maximum flow of data to or from any single processor. `TotalV` assumes that by reducing network contention and the total number of elements moved, the remapping time will be reduced. Both an optimal and a heuristic greedy algorithm have been implemented for solving the processor reassignment problem using `TotalV` [12]. Applying the heuristic procedure to the similarity matrix in Fig. 2.1 generates the processor assignment shown in the bottom row. It was proved in [12] that a processor assignment obtained using the heuristic algorithm can never result in a data movement cost that is twice that of the optimal assignment. `MaxV`, on the other hand, considers data redistribution in terms of solving a load imbalance problem, where it is more important to minimize the workload of the most heavily-weighted processor than to minimize the sum of all the loads. An optimal algorithm for solving the assignment problem using `MaxV` has also been implemented [12].

2.4. Cost models. 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 small individual 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. This remapping procedure closely follows the superstep model of BSP [18].

In [1], we derived the expected time for the redistribution procedure on bandwidth-rich systems as:

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

where $\texttt{MaxSR} = \max(\texttt{ElemsSent}) + \max(\texttt{ElemsRecd})$ and can be quickly derived from the solved similarity matrix S; γ represents the total computation and communication cost to process each redistributed element; and O is the predicted sum of all constant overheads including the cost of processing partition boundary information, data compaction costs, communication start-up costs, and barrier synchronizations. In order to compute the slope and intercept of this linear function, several data points need to be generated for various redistribution patterns and their corresponding run times. A simple least squares fit can then be used to approximate γ and O. This procedure needs to be performed only once for each architecture, and the values of γ and O can then be used in actual computations to estimate the redistribution cost. Note that there is a close relationship between $\texttt{MaxSR}$ of the remapping cost model and the theoretical metric $\texttt{MaxV}$. The optimal similarity matrix solution for $\texttt{MaxSR}$ is provably no more than twice that of $\texttt{MaxV}$.

The computational gain due to repartitioning is proportional to the decrease in the load imbalance achieved by running the adapted mesh on the new partitions rather than on the old partitions. It can be expressed as $T_{\rm iter}N_{\rm adapt}(W_{\rm max}^{\rm old} - W_{\rm max}^{\rm new})$, where $T_{\rm iter}$ is the time required to run one solver iteration on one element of the original mesh, $N_{\rm adapt}$ is the number of solver iterations between mesh adaptions, and $W_{\rm max}^{\rm old}$ and $W_{\rm max}^{\rm new}$ are the sum of the $w_{\rm comp}$ on the most heavily-loaded processor for the old and new partitionings, respectively.

An additional benefit of data redistribution before mesh subdivision is the improved performance of the refinement procedure, which runs in a more load balanced fashion. The savings is therefore incorporated as an additional term in the computational gain expression. The new partitioning and mapping are accepted if the computational gain is larger than the redistribution cost:

$$T_{\mathrm{iter}} N_{\mathrm{adapt}} (W^{\mathrm{old}}_{\max} - W^{\mathrm{new}}_{\max}) + T_{\mathsf{refine}} \left(\frac{W^{\mathrm{new}}_{\max}}{W^{\mathrm{old}}_{\max}} - 1 \right) > \gamma \times \mathtt{MaxSR} + O,$$

where T_{refine} is the time required to perform the subdivision phase based on the edge-marking patterns. In that case, all data is appropriately redistributed.

3. Experimental results. The 3D_TAG parallel mesh adaption procedure and the PLUM global load balancing strategy have been implemented in C and C++, with the parallel activities in MPI for portability. No architecture-specific optimizations were used to obtain the performance results reported in this paper.

All experiments were performed on a wide-node IBM SP2 and a SGI/ Cray Origin2000. The SP2 is located in the Numerical Aerospace Simulation division at NASA Ames Research Center. It consists of RS6000/590 processors, which are connected through a high performance switch, called the Vulcan chip. Each chip connects up to eight processors, and eight Vulcan chips comprise a switching board. An advantage of this interconnection mechanism and wormhole routing is that all nodes can be considered equidistant from one another.

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.

3.1. Helicopter rotor test case. The computational mesh used for the first set of experiments is one used to simulate an acoustics wind-tunnel test [14]. In that experiment, a 1/7th-scale model of a UH-1H helicopter rotor blade was tested over a range of subsonic and transonic hover-tip Mach numbers. Detailed numerical results of the simulation are given in [17]. In this paper, results are presented for one refinement step where edges are targeted for subdivision based on an error indicator [17] calculated directly from the flow solution. Three different cases are studied with varying fractions of the domain being targeted for refinement. The strategies, called Real_1, Real_2, and Real_3, subdivided 5%, 33%, and 60% of the 78,343 edges of initial mesh. Table 3.1 lists the grid sizes for this single level of refinement for each of the three cases.

TABLE 3.1
Grid sizes for the three different refinement strategies

	Vertices	Elements	Edges
Initial	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

3.1.1. PLUM on SP2 and Origin2000. Figure 3.1 illustrates the parallel speedup for each of the three edge-marking strategies on the SP2 and the Origin2000. Two sets of results are presented for each machine: one when data remapping is performed after mesh refinement, and the other when remapping is performed before refinement. The speedup numbers are almost identical on the two machines. The Real_3 case shows the best speedup performance because it is the most computation intensive. Remapping the data before refinement has the largest relative effect for Real_1, 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_3 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 an SP2 is given in [13].

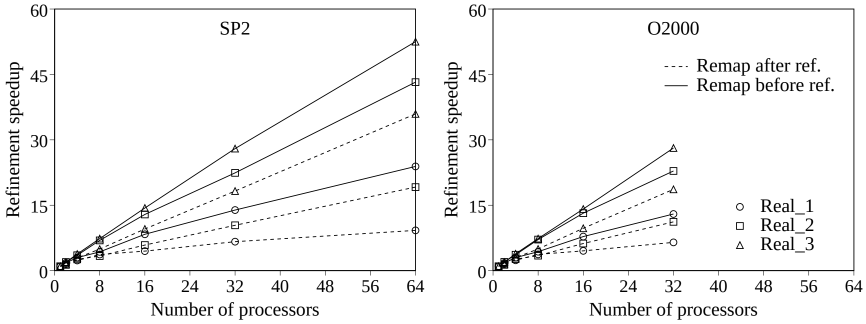


FIG. 3.1. *Speedup of* 3D_TAG *on the SP2 and the Origin2000 when data is remapped either after or before mesh refinement.*

To compare the performance on the SP2 and the Origin2000 more critically, one needs to look at the actual mesh adaption times rather than the speedup values. These results are presented in Table 3.2 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

Table 3.2
Execution time of **3D_TAG** *on the SP2 and the Origin2000 when data is remapped before mesh refinement*

	Real_1		Real_2		Real_3	
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	

adaption code does not use the floating-point units on the SP2, thereby adversely affecting its overall performance.

Figure 3.2 shows the remapping time for each of the three cases on the SP2 and the Origin2000. As in Fig. 3.1, results are presented both when the data remapping is done after and before the mesh subdivision. 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 the mesh grows in size only after the data has been redistributed. The remapping times also decrease as the number of processors is increased. This is because even though the total volume of data movement increases with the number of processors, there are actually more processors to share the work. The remapping times when data is moved before mesh refinement are reproduced in Table 3.3 since the exact values are difficult to read off the log-scale in Fig. 3.2.

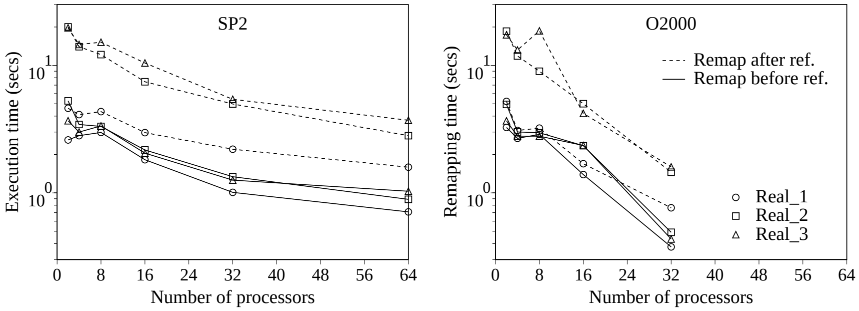


Fig. 3.2. *Remapping time within* **PLUM** *on the SP2 and the Origin2000 when data is redistributed either after or before mesh refinement.*

Perhaps the most remarkable feature of these results is the dramatic re-

TABLE 3.3
Remapping time within **PLUM** *on the SP2 and the Origin2000 when data is redistributed before mesh refinement*

	Real_1		Real_2		Real_3	
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	

duction 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 upto 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 3.2 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. 3.1 and 3.2 demonstrate that our methodology within PLUM is effective in significantly reducing the data remapping time and improving the parallel performance of mesh refinement.

Figure 3.3 shows how the execution time is spent during the refinement and the subsequent load balancing phases for the three different cases on the SP2 and the Origin2000. The processor reassignment times are not shown since they are negligible compared to the other times. Note that the graphs for the two machines have different scales on the axes; however, both machines show similar qualitative behavior. The repartitioning curves, using PMeTiS [10], are almost identical for the three cases on each machine 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

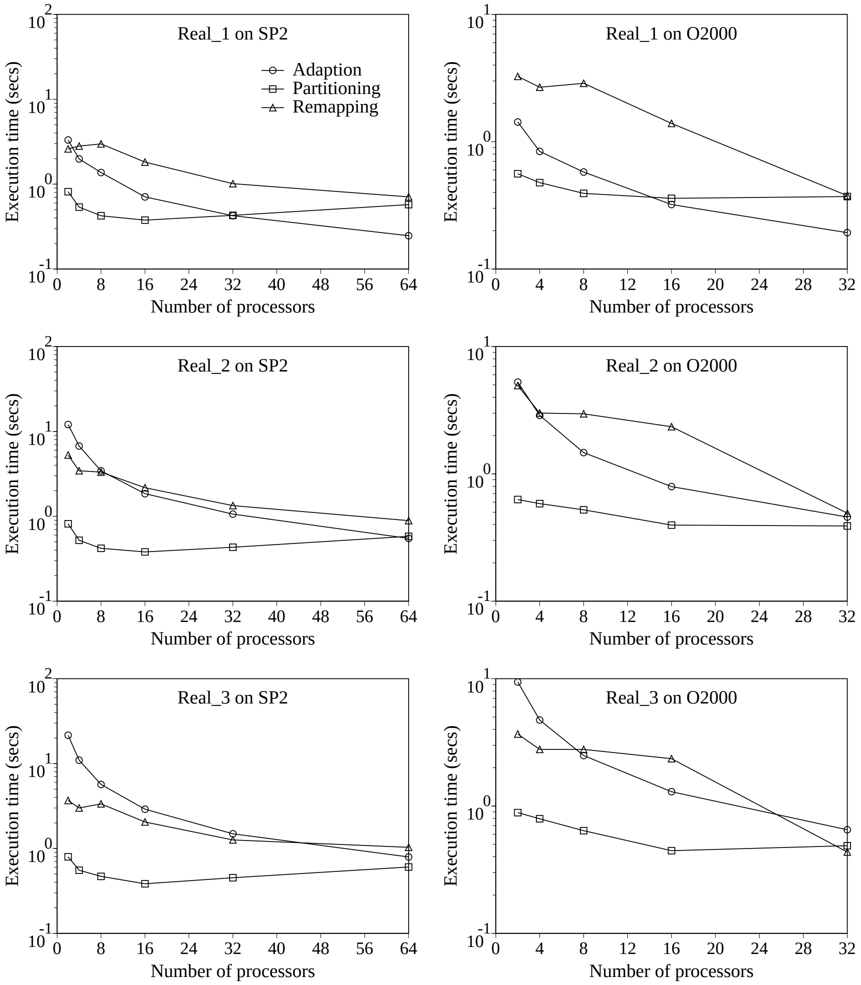


FIG. 3.3. *Anatomy of execution times for the* **Real_1**, **Real_2**, *and* **Real_3** *refinement strategies on the SP2 and the Origin2000.*

larger initial mesh, the minimum partitioning time will occur for a higher number of processors. These results show that PLUM can be successfully ported to different platforms without any code modifications.

3.1.2. Deriving the redistribution cost model. It is important to note from the results in Fig. 3.3 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 in dynamic load balancing problems. It is 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 next set of experiments is performed to compute the slope γ and the intercept O of our redistribution cost model on both the SP2 and the Origin2000. Experimental data is gathered by running various redistribution patterns. The remapping times are then plotted against two metrics, `TotalV` and `MaxSR`, in Fig. 3.4. Results demonstrate that on an SP2, 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 redistribution time. These results indicate that, on the SP2, our redistribution model successfully estimates the data remapping time, and that reducing the bottleneck, rather than the aggregate, overhead guarantees a reduction in the redistribution time.

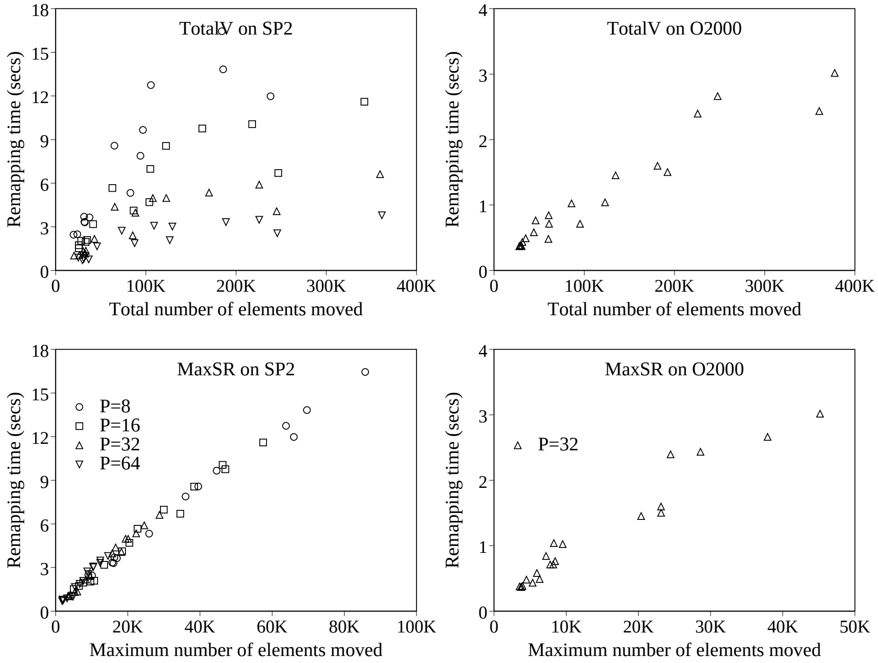


FIG. 3.4. *Remapping time as a function of* `TotalV` *and* `MaxSR` *on the SP2 and the Origin2000.*

The situation is quite different on the Origin2000. Remapping times were extremely unpredictable for $P < 32$; hence, they are not shown in Fig. 3.4. 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.

3.2. 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. 3.5. The test case is generated by refining all elements within a cylindrical volume moving left to right across the domain with constant velocity, while coarsening previously-refined elements in its wake. The performance of PLUM is 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.

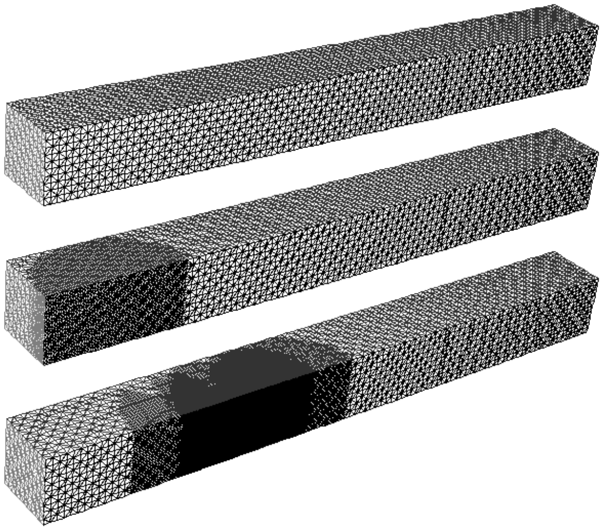


FIG. 3.5. *Initial and adapted meshes (after levels 1 and 5) for the simulated unsteady experiment.*

Figure 3.6 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. 3.5, contains 50,000 tetrahedral elements. The mesh after the first and fifth adaptions for Sequence_1 are also shown in Fig. 3.5. 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

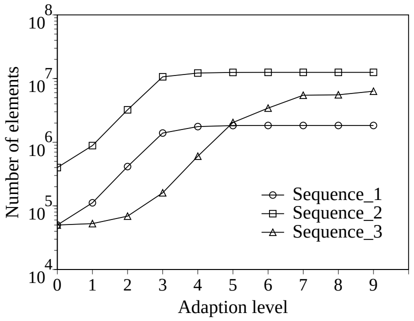


FIG. 3.6. *Progression of grid sizes through nine levels of adaption for the unsteady simulation.*

refinement is almost exactly compensated by mesh coarsening. A third scenario, called Sequence_3, was 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.

3.2.1. Comparison of partitioners. Table 3.4 presents the partitioning times for Sequence_1 using the five different partitioners briefly described in Section 2.2. PMeTiS is the parallel multilevel k-way partitioning scheme of Karypis and Kumar [10], UAMeTiS and DAMeTiS are multilevel undirected and directed repartitioning algorithms of Schloegel, Karypis, and Kumar [15], and Jostle-MS and Jostle-MD are multilevel-static and multilevel-dynamic configurations of the Jostle partitioner of Walshaw, Cross, and Everett [21]. 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.

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 3.5 and 3.6. PMeTiS does an excellent job of consistently reducing the load imbalance factor to within 6% of ideal (cf. Table 3.5). The Jostle partitioners are only slightly worse than PMeTiS,

[1] The last row in Tables 3.4–3.11 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 3.4
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

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.

TABLE 3.5
Load imbalance factor before and after mesh partitioning for P=64 using a variety of partitioners for **Sequence_1**

L	PMeTiS		UAMeTiS		DAMeTiS		Jostle-MS		Jostle-MD	
	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 3.6). 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

TABLE 3.6
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

since the load imbalance factor has to be reduced.

Our overall conclusions from the results presented in Tables 3.4–3.6 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 3.7. 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 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

TABLE 3.7

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

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 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.

3.2.2. 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 pared 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 3.8. Consistent with the results in Table 3.4, DAMeTiS is slightly faster than PMeTiS on both machines. Consistent with the results in Table 3.2, 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 3.7. Finally, the remapping times are about three times faster on the Origin2000 than on the SP2 as was also shown earlier in Table 3.3.

The quality of load balancing for this experimental case is presented in

TABLE 3.8

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

Table 3.9. 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 3.9 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 3.5 and 3.6; however, the

TABLE 3.9

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

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.

3.2.3. Coarse vs. fine initial mesh. Figure 3.7 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 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.

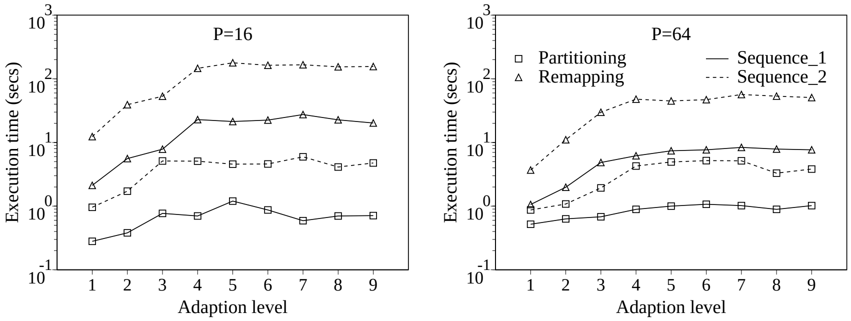


FIG. 3.7. *PMeTiS partitioning and remapping times using the heuristic strategy for P=16 and 64 on an SP2 for* **Sequence_1** *and* **Sequence_2**.

Figure 3.8 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. 3.7). The percentage of cut edges always increases with the number of processors. This is expected since the surface-to-volume ratio increases with the number of partitions. 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

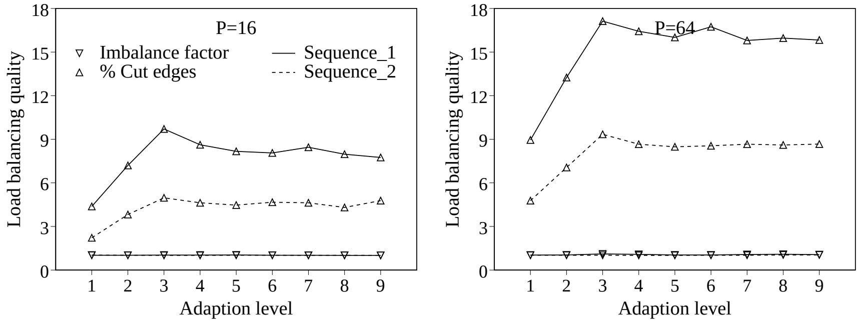


FIG. 3.8. *Load imbalance factor and percentage of cut edges after mesh partitioning using PMeTiS for P=16 and 64 for* Sequence_1 *and* Sequence_2*. Note that the imbalance factor curves for the two sequences are overlaid.*

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.

3.2.4. 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 3.10. A comparison with the results in Table 3.8 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.

The quality of load balancing is presented in Table 3.11. PMeTiS is once again significantly better than DAMeTiS in terms of the load imbalance factor. Compared to the corresponding results in Table 3.9, 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, lim-

TABLE 3.10
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

its 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 3.9. 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

TABLE 3.11
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

adjustable vertex and edge weights can be successfully used even when the mesh is growing significantly and rapidly.

4. Conclusions. We have shown in this paper that our load balancing scheme, called PLUM, works well for both steady and unsteady adaptive problems with many levels of adaption, even when using a coarse initial mesh. A finer starting mesh may be used to achieve lower edge cuts and marginally better load balance, but is generally not worth the increased partitioning and data remapping times. Portability was demonstrated by presenting results on the two vastly different architectures of the SP2 and the Origin2000, without the need for any code modifications. We examined the performance of five state-of-the-art parallel partitioners within PLUM, and found that a global repartitioner can outperform diffusive schemes in both subdomain quality and remapping overhead. 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. Finally, we 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.

Acknowledgements. The authors would like to sincerely thank Vipin Kumar and George Karypis at the University of Minnesota for their help with the MeTiS partitioners, and Chris Walshaw at the University of Greenwich for his help with the Jostle partitioners.

REFERENCES

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

[2] R. Biswas, L. Oliker, and A. Sohn, *Global load balancing with parallel mesh adaption on distributed-memory systems*, Supercomputing (1996).

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

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

[5] H.L. de Cougny, K.D. Devine, J.E. Flaherty, R.M. Loy, C. Ozturan, and M.S. Shephard, *Load balancing for the parallel adaptive solution of partial differential equations*, Applied Numerical Mathematics, 16 (1994), pp. 157–182.

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

[7] 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 (1997), pp. 360–367.

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

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

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

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

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

[13] 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 (1996), pp. 35–47.

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

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

[16] 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 (1996), pp. 26–33.

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

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

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

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

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