RIACS

NASA/CR- -1998 208235

Research Institute for Advanced Computer Science
NASA Ames Research Center

NAS2-96027
226787
IN-61

Performance Analysis and Portability of the PLUM Load Balancing System

Leonid Oliker, Rupak Biswas, and Harold N. Gabow

RIACS Technical Report 98.02 May 1998

To appear in the *Proceedings of the Fourth International Euro-Par Conference*
University of Southampton, UK, September 1-4, 1998.

Performance Analysis and Portability of the PLUM Load Balancing System

Leonid Oliker[1], Rupak Biswas[2], and Harold N. Gabow[3]

[1] RIACS, NASA Ames Research Center, Moffett Field, CA 94035, USA
[2] MRJ, NASA Ames Research Center, Moffett Field, CA 94035, USA
[3] CS Department, University of Colorado, Boulder, CO 80309, USA

Abstract. The ability to dynamically adapt an unstructured mesh is a powerful tool for solving computational problems with evolving physical features; however, an efficient parallel implementation is rather difficult. To address this problem, we have developed PLUM, an automatic portable framework for performing adaptive numerical computations in a message-passing environment. PLUM requires that all data be globally redistributed after each mesh adaption to achieve load balance. We present an algorithm for minimizing this remapping overhead by guaranteeing an optimal processor reassignment. We also show that the data redistribution cost can be significantly reduced by applying our heuristic processor reassignment algorithm to the default mapping of the parallel partitioner. Portability is examined by comparing performance on a SP2, an Origin2000, and a T3E. Results show that PLUM can be successfully ported to different platforms without any code modifications.

1 Introduction

The ability to dynamically adapt an unstructured mesh is a powerful tool for efficiently solving computational problems with evolving physical features. Standard fixed-mesh numerical methods can be made more cost-effective by locally refining and coarsening the mesh to capture these phenomena of interest. Unfortunately, an efficient parallelization of these adaptive methods is rather difficult, primarily due to the load imbalance created by the dynamically-changing nonuniform grid. Nonetheless, it is generally thought that unstructured adaptive-grid techniques will constitute a significant fraction of future high-performance supercomputing.

With this goal in mind, we have developed a novel method, called PLUM [7], that dynamically balances processor workloads with a global view when performing adaptive numerical calculations in a parallel message-passing environment. The mesh is first partitioned and mapped among the available processors. Once an acceptable numerical solution is obtained, the mesh adaption procedure [8] is invoked. Mesh edges are targeted for coarsening or refinement based on an error indicator computed from the solution. The old mesh is then coarsened, resulting in a smaller grid. Since edges have already been marked for refinement, the new mesh can be exactly predicted before actually performing the refinement step. Program control is thus passed to the load balancer at this time. If the current partitions will become load imbalanced after adaption, a repartitioner is used to divide the new mesh into subgrids. The new partitions are then reassigned among 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.

2 Dynamic Load Balancing

2.1 Repartitioning the Initial Mesh Dual Graph

Repeatedly using the dual of the initial computational mesh for dynamic load balancing is one of the key features of PLUM [7]. Each dual graph vertex has a computational weight, w_{comp}, and a remapping weight, w_{remap}. These weights model the processing workload and the cost of moving the corresponding element from one processor to another. Every dual graph edge also has a weight, w_{comm}, that models the runtime communication. New computational grids obtained by adaption are represented by modifying these three weights. If the dual graph with a new set of w_{comp} is deemed unbalanced, the mesh is repartitioned.

2.2 Processor Reassignment

New partitions generated by a partitioner are mapped to 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, and each processor is assigned F partitions. Allowing multiple partitions per processor reduces the volume of data movement but increases the partitioning and reassignment times [7].

We first generate 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 M_{ij} is the sum of the w_{remap} values of all the dual graph vertices in new partition j that already reside on processor i. Various cost functions are usually needed to solve the processor reassignment problem using M for different machine architectures. We present three general metrics: `TotalV`, `MaxV`, and `MaxSR`, which model the remapping cost on most multiprocessor systems. `TotalV` minimizes the total volume of data moved among all the processors, `MaxV` minimizes the maximum flow of data to *or* from any single processor, while `MaxSR` minimizes the sum of the maximum flow of data to *and* from any processor. Experimental results [2] have indicated the usefulness of these metrics in predicting the actual remapping costs. A greedy heuristic algorithm to minimize the remapping overhead is also presented.

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

$$\mathcal{F} = \sum_{i=1}^{P} \sum_{f=1}^{F} M_{ij_{i\_f}}$$

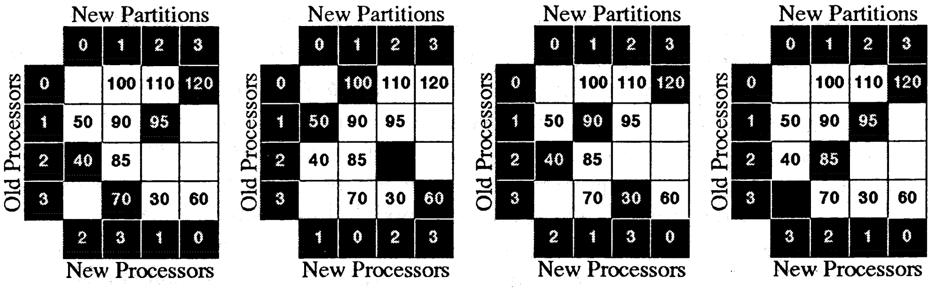
is maximized subject to the constraint

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

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. 1(a).

The fastest MWBG algorithm can compute a matching in $O(|V|^2 \log |V| + |V||E|)$ time [3], or in $O(|V|^{1/2}|E| \log(|V|C))$ time if all edge costs are integers of absolute value at most C [5]. 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.



(a)	(b)	(c)	(d)
TotalV moved = 525	TotalV moved = 640	TotalV moved = 570	TotalV moved = 550
MaxV moved = 275	MaxV moved = 245	MaxV moved = 255	MaxV moved = 260
MaxSR moved = 485	MaxSR moved = 475	MaxSR moved = 465	MaxSR moved = 470

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

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 $\max\big(\alpha \times \max(\texttt{ElemsSent}), \beta \times \max(\texttt{ElemsRecd})\big)$, 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:

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

Currently, our framework for the `MaxV` metric is restricted to $F = 1$.

We have implemented the BMCM algorithm of Bhat [1] 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 [4], has a runtime of $O((|V| \log |V|)^{1/2}|E|)$.

The new processor assignment for the similarity matrix in Fig. 1 using this approach with $\alpha = \beta = 1$ is shown in Fig. 1(b). Notice that the total number of elements moved in Fig. 1(b) is larger than the corresponding value in Fig. 1(a); however, the maximum number of elements moved is smaller.

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 $\big(\alpha \times \max(\texttt{ElemsSent})$ $+ \beta \times \max(\texttt{ElemsRecd})\big)$. 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*) corresponding to the total cost of sending and receiving data, when partition j is mapped to processor i:

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

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 σ_i+ $\mathtt{MaxV}(G_i)$. Defining

$$\mathtt{MaxSR}(i) = \min_{i_{\min} \leq j \leq i} (\sigma_j + \mathtt{MaxV}(G_j)),$$

it is easy to see that $\mathtt{MaxSR}(k)$ 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 $\mathtt{MaxSR}(k)$. However, we can prematurely terminate the algorithm if there exists an $i_{\max}$ such that $\sigma_{i_{\max}+1} \geq \mathtt{MaxSR}(i_{\max})$, since it is then guaranteed that the MaxSR solution is $\mathtt{MaxSR}(i_{\max})$.

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 $\mathtt{MaxV}(G_i)$ deleted. Take the matching $\mathcal{M}$ on G, and delete all unmatched edges of weight $\mathtt{MaxV}(G_i)$. 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 $\mathtt{MaxV}(G_i) = \mathtt{MaxV}(G_{i+1})$. 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 $\mathtt{MaxV}(G_{i+1})$. At this point our algorithm to solve the BMCM problem of G_{i+1} will stop, since no augmenting path will be found.

To see that this algorithm runs in $O(|E|^2)$, note that each search for an augmenting path uses time $O(|E|)$ and that there are $O(|E|)$ such searches. A successful search for an augmenting path for edge (u, v) permanently eliminates it 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. 1 using the DBMCM algorithm with $\alpha = \beta = 1$ is shown in Fig. 1(c). Notice that the MaxSR solution is minimized; however, the number of TotalV elements moved is larger than the corresponding value in Fig. 1(a), and more MaxV elements are moved than in Fig. 1(b). Also note that the optimal similarity matrix solution for MaxSR is provably no more than twice that of MaxV.

Heuristic Algorithm. We have developed a heuristic greedy algorithm that gives a suboptimal solution to the TotalV metric in $O(|E|)$ steps [7]. All partitions are initially 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. Oliker and Biswas [7] proved 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. In addition, experimental results in Sec. 3.1 demonstrate that our heuristic quickly finds high quality solutions for all three metrics. Applying this heuristic algorithm to the similarity matrix in Fig. 1 generates the new processor assignment shown in Fig. 1(d).

2.3 Remapping Cost Model

Once the reassignment problem is solved, a model is needed to quickly predict the expected redistribution cost for a given architecture. 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 distributes the data to its destination; and finally, the received data is integrated into each partition and the boundary information is consistently updated. This remapping procedure closely follows the superstep model of BSP [9].

The expected time for the redistribution procedure on bandwidth-rich systems can be expressed as $\gamma \times \texttt{MaxSR} + O$, where $\texttt{MaxSR} = \max(\texttt{ElemsSent}) + \max(\texttt{ElemsRecd})$, γ is the total computation and communication cost to process each redistributed element, and O is the sum of all constant overheads [7]. This formulation demonstrates the 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. To compute γ and O, a simple least squares fit through several data points for various redistribution patterns and their corresponding runtimes can be used. 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.

3 Experimental Results

The 3D_TAG parallel mesh adaption procedure [8] and the PLUM global load balancing strategy [7] have been implemented in C and C++, with the parallel activities in MPI for portability. All experiments were performed on the wide-node SP2 at NASA Ames, the Origin2000 at NCSA, and the T3E at NASA Goddard, without any machine-specific optimizations.

The computational mesh used in this paper is one used to simulate an acoustics wind-tunnel experiment of a UH-1H helicopter rotor blade [7]. Three different cases are studied, with varying fractions of the domain targeted for refinement based on an error indicator calculated directly from the flow solution. The strategies, called Real_1, Real_2, and Real_3, subdivided 5%, 33%, and 60% of the 78,343 edges of the initial mesh. This increased the number of mesh elements from 60,968 to 82,489, 201,780, and 321,841, respectively.

3.1 Comparison of Reassignment Algorithms

Table 1 presents a comparison of our five different processor reassignment algorithms in terms of the reassignment time (in secs) and the amount of data movement. Results are shown for the Real_2 strategy on the SP2 with $F = 1$. The PMeTiS [6] case does not require any explicit processor reassignment since we choose the default partition-to-processor mapping given by the partitioner. The poor performance for all three metrics is expected since PMeTiS is a global partitioner that does not attempt to minimize the remapping overhead. Previous work [2] compared the performance of PMeTiS with other partitioners.

Table 1. Comparison of reassignment algorithms for Real_2 on the SP2 with $F = 1$

	P = 32				P = 64			
	TotalV	MaxV	MaxSR	Reass.	TotalV	MaxV	MaxSR	Reass.
Algthm.	Metric	Metric	Metric	Time	Metric	Metric	Metric	Time
PMeTiS	58297	5067	7467	0.0000	67439	2667	4452	0.0000
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
Heuristic	35032	4410	5809	0.0017	38283	2261	3123	0.0088

The execution times of the other four algorithms increase with the number of processors because of the growth in the size of the similarity matrix; however, the heuristic time for 64 processors is still very small and acceptable. The total volume of data movement is obviously the 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 all the other algorithms give almost identical results for the MaxV metric. In our helicopter rotor experiment, only a few localized regions of the domain incur a dramatic increase in the number of 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, 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 architectures, computing its optimal solution can provide useful information. Notice that the minimum TotalV increases slightly as P grows from 32 to 64, while MaxSR is dramatically reduced by over 45%. This trend continues as the number of processors increases, and indicates that PLUM will remain viable on a large number of processors, since the per processor workload 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. Similar results were obtained for the other edge-marking strategies.

3.2 Portability Analysis

The top three plots in Fig. 2 illustrate parallel speedup for the three edge-marking strategies on the SP2, Origin2000, and T3E. Two sets of results are presented for each machine: one when data remapping is performed after mesh refinement, and the other when remapping is done before refinement. The speedup numbers are almost identical on all three machines. The Real_3 case shows the best speedup values because it is the most computation intensive. Remapping data before refinement has the largest relative effect for Real_1, because it has the smallest refinement region and predictively load balancing the refined mesh returns the biggest benefit. The best results are for Real_3 with remapping before refinement, showing an efficiency greater than 87% on 32 processors.

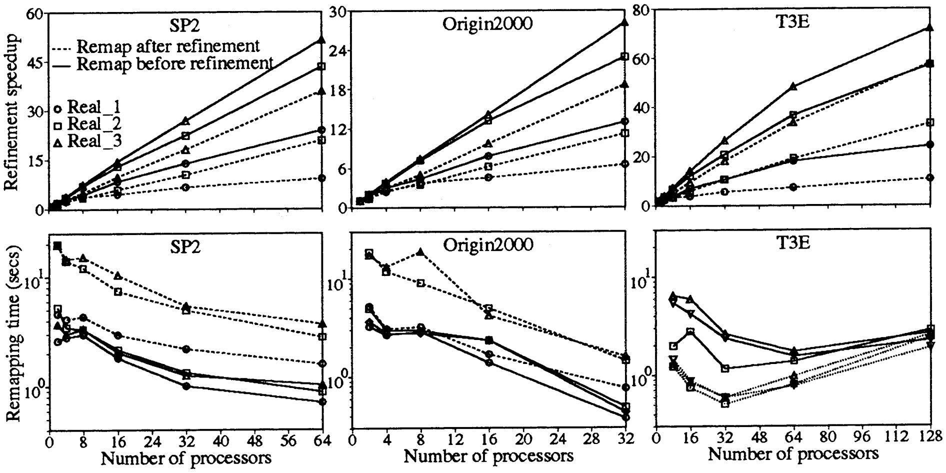


Fig. 2. Refinement speedup (top) and remapping time (bottom) within PLUM on the SP2, Origin2000, and T3E, when data is redistributed after or before mesh refinement

To compare the performance on the SP2, Origin2000, and T3E more critically, one needs to look at the actual times rather than the speedup values. Table 2 shows how the execution time (in secs) is spent during the refinement and subsequent load balancing phases for the Real_2 case when data is remapped before the subdivision phase. The processor reassignment times are not presented since they are negligible compared to other times, as is evident from Table 1. Notice that the T3E adaption times are consistently more than 1.4 times faster than the Origin2000 and three times faster than the SP2. One reason for this performance difference is the disparity in the clock speeds of the three machines.

Table 2. Anatomy of execution times for Real_2 on the Origin2000, SP2, and T3E

	Adaption Time			Remapping Time			Partitioning Time		
P	O2000	SP2	T3E	O2000	SP2	T3E	O2000	SP2	T3E
2	5.261	12.06	3.455	3.005	3.440	2.648	0.628	0.815	0.701
4	2.880	6.734	1.956	3.005	3.440	1.501	0.584	0.537	0.477
8	1.470	3.434	1.034	2.963	3.321	1.449	0.522	0.424	0.359
16	0.794	1.846	0.568	2.346	2.173	0.880	0.396	0.377	0.301
32	0.458	1.061	0.333	0.491	1.338	0.592	0.389	0.429	0.302
64		0.550	0.188		0.890	0.778		0.574	0.425
128			0.121			1.894			0.599

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

The bottom three plots in Fig. 2 show the remapping time for each of the three cases on the SP2, Origin2000, and T3E. In almost every case, 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. In general, 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 for the Real_2 case in Table 2 since the exact values are difficult to read off the log-scale.

Perhaps the most remarkable feature of these results is the peculiar behavior of the T3E when $P \geq 64$. When using up to 32 processors, the remapping performance of the T3E is very similar to that of the SP2 and Origin2000. It closely follows the redistribution cost model given in Sec. 2.3, and achieves a significant runtime improvement when remapping is performed prior to refinement. However, for 64 and 128 processors, the remapping overhead on the T3E begins to increase and violates our cost model. The runtime difference when data is remapped before and after refinement is dramatically diminished; in fact, all the remapping times begin to converge to a single value! This indicates that the remapping time is no longer affected only by the volume of data redistributed but also by the interprocessor communication pattern. One way of potentially improving these results is to take advantage of the T3E's ability to efficiently perform one-sided communication.

Another surprising result is the dramatic reduction in remapping times when using 32 processors on the Origin2000. This is probably because network contention with other jobs is essentially removed when using the entire machine. When using up to 16 processors, the remapping times on the SP2 and the Origin2000 are comparable, while the T3E is about twice as fast. Recall that the remapping phase within PLUM consists of both communication and computation. Since the results in Table 2 indicate that computation is faster on the Origin2000, it is reasonable to infer that bulk communication is faster on the SP2. These results generally demonstrate that our methodology within PLUM is effective in significantly reducing the data remapping time and improving the parallel performance of mesh refinement.

Table 2 also presents the PMeTiS partitioning times for Real_2 on all three systems; the results for Real_1 and Real_3 are almost identical because the time to repartition mostly depends on the initial problem size. There is, however, some dependence on the number of processors used. 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. Table 2 demonstrates that PMeTiS is fast enough to be effectively used within our framework, and that PLUM can be successfully ported to different platforms without any code modifications.

4 Conclusions

In this paper, we verified the effectiveness of our PLUM load balancer for adaptive unstructured meshes on a helicopter acoustics problem. We developed three generic metrics to model the remapping cost on most multiprocessor systems. Optimal solutions for these metrics, as well as a heuristic approach were implemented. We showed that the data redistribution overhead can be significantly reduced by applying our heuristic processor reassignment algorithm to the default mapping given by the global partitioner. Portability was demonstrated by presenting results on the three vastly different architectures of the SP2, Origin2000, and T3E, without the need for any code modifications. Results showed that, in general, PLUM will remain viable on large numbers of processors. However, our redistribution cost model was violated on the T3E when 64 or more processors were used. Future research will address the improvement of these results, and the development of a more comprehensive remapping cost model.

References

1. Bhat, K.: An $O(n^{2.5}\log_2 n)$ time algorithm for the bottleneck assignment problems. AT&T Bell Laboratories Unpublished Report (1984)
2. Biswas, R., Oliker, L.: Experiments with repartitioning and load balancing adaptive meshes. NASA Ames Research Center Technical Report NAS-97-021 (1997)
3. Fredman, M., Tarjan, R.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34** (1987) 596–615
4. Gabow, H., Tarjan, R.: Algorithms for two bottleneck optimization problems. J. of Alg. **9** (1988) 411–417
5. Gabow, H., Tarjan, R.: Faster scaling algorithms for network problems. SIAM J. on Comput. **18** (1989) 1013–1036
6. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning scheme for irregular graphs. University of Minnesota Technical Report 96-036 (1996)
7. Oliker, L., Biswas, R.: PLUM: Parallel load balancing for adaptive unstructured meshes. NASA Ames Research Center Technical Report NAS-97-020 (1997)
8. Oliker, L., Biswas, R., Strawn, R.: Parallel implementation of an adaptive scheme for 3D unstructured grids on the SP2. Springer-Verlag LNCS **1117** (1996) 35–47
9. Valiant, L.: A bridging model for parallel computation. Comm. ACM **33** (1990) 103–111

User: oliker

Date: Tue May 11 14:04:36 2000

For:	oliker
Date:	Tue May 12 12:13:46 PDT 1998
Creator:	dvipsk 5.66a Copyright 1986-97 Radical Eye Software (www.radicaleye.com)

Job No:	155
Emulation:	postscript
Submit queue:	IF 1 / Ethernet / UHS
Submitted:	Tue May 11 14:04:36 2000
Started:	Tue May 11 14:04:36 2000

QMS 3225 Print System **QMS3**

User: oliker

Date: Wed Apr 28 19:14:55 2000

For:	oliker
Date:	Wed Apr 29 17:31:19 PDT 1998
Job No:	159
Emulation:	pcl5
Submit queue:	IF 1 / Ethernet / UHS
Submitted:	Wed Apr 28 19:14:55 2000
Started:	Wed Apr 28 19:14:55 2000

QMS 3225 Print System **QMS3**

QMS