Portable Parallel Programming for the Dynamic Load Balancing of Unstructured Grid Applications

Rupak Biswas
MRJ Technology Solutions
NASA Ames Research Center
Moffett Field, CA 94035, USA
rbiswas@nas.nasa.gov

Sajal K. Das, Daniel Harvey
Dept of Computer Sciences
University of North Texas
Denton, TX 76203, USA
{das,harvey}@cs.unt.edu

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

Abstract

The ability to dynamically adapt an unstructured grid (or mesh) is a powerful tool for solving computational problems with evolving physical features; however, an efficient parallel implementation is rather difficult, particularly from the viewpoint of portability on various multiprocessor platforms. We address this problem by developing PLUM, an automatic and architecture-independent framework for adaptive numerical computations in a message-passing environment. Portability is demonstrated by comparing performance on an SP2, an Origin2000, and a T3E, without any code modifications. We also present a general-purpose load balancer that utilizes symmetric broadcast networks (SBN) as the underlying communication pattern, with a goal to providing a global view of system loads across processors. Experiments on an SP2 and an Origin2000 demonstrate the portability of our approach which achieves superb load balance at the cost of minimal extra overhead.

1. Introduction

The success of parallel computing in solving real-life, computation-intensive problems relies on their efficient mapping and execution on commercially available multiprocessor architectures. When the algorithms and data structures corresponding to these problems are *dynamic* in nature (i.e., their computational workloads grow or shrink at runtime) or are intrinsically *unstructured*, mapping them on to distributed-memory parallel machines with dynamic load balancing offers considerable challenges. Dynamic load balancing aims to balance processor workloads at runtime while minimizing inter-processor communication. With the proliferation of parallel computing, dynamic load balancing has become extremely important in several applications like scientific computing, task scheduling, sparse matrix computations, parallel discrete event simulation, and data mining.

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 adaptive methods is rather difficult, primarily due to the load imbalance created by the dynamically-changing nonuniform grids. Nonetheless, it is believed that unstructured adaptive-grid techniques will constitute a significant fraction of future high-performance supercomputing.

As an example, if a full-scale problem in computational fluid dynamics were to be solved efficiently in parallel, dynamic mesh adaptation would cause load imbalance among processors. This, in turn, would require large amounts of data movement at runtime. It is therefore imperative to have an efficient dynamic load balancing mechanism as part of the solution procedure. However, since the computational mesh will be frequently adapted for unsteady flows, the runtime load also has to be balanced at each step. In other words, the dynamic load balancing procedure itself must not pose a major overhead. This motivates our work.

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. Examining the performance of PLUM for an actual workload, which simulates an acoustic wind-tunnel experiment of a helicopter rotor blade, on three different parallel machines demonstrates that it can be successfully ported without any code modifications.

We propose another new approach to dynamic load balancing for unstructured grid applications based on defining a robust communication pattern (logical or physical) among processors, called *symmetric broadcast networks* (SBN) [3]. It is adaptive and decentralized in nature, and can be ported to any topological architecture through efficient embedding techniques. Portability results for an adaptive unsteady grid workload, generated by propagating a simulated shock wave through a tube, show that our approach reduces the redistribution cost at the expense of a minimal extra communication overhead. In many mesh adaptation applications in which the data redistribution cost dominates the processing and communication cost, this is an acceptable trade-off.

2. Architecture-Independent Load Balancer

PLUM is an automatic and portable load balancing environment, specifically created to handle adaptive unstructured grid applications. It differs from most other load balancers in that it dynamically balances processor workloads with a *global* view [1, 7]. In this paper, we examine its architecture-independent feature by comparing results for a test case running on an SP2, Origin2000, and T3E.

PLUM consists of a partitioner and a remapper that load balance and redistribute the computational mesh when necessary. After an initial partitioning and mapping of the unstructured mesh, a solver executes several iterations of the application. A mesh adaptation procedure is invoked when the mesh is desired to be refined or coarsened. PLUM then gains control to determine if the workload among the processors has become unbalanced due to the mesh adaptation, and to take appropriate action. If load balancing is required, the adapted mesh is repartitioned and reassigned among the processors so that the cost of data movement is minimized. If the estimated remapping cost is lower than the expected computational gain to be achieved, the grid is remapped among the processors before solver execution is resumed.

Extensive details about PLUM are given in [7]. For completeness, we enumerate some of its salient features.

- **Reusing the initial dual graph:** PLUM repeatedly utilizes the dual of the initial mesh for the purposes of load balancing. This keeps the complexity of the partitioning and reassignment phases constant during the course of an adaptive computation. New computational grids obtained by adaptation are translated by changing the *weights* of the vertices and edges of the dual graph.
- **Parallel mesh repartitioning:** PLUM can use any general-purpose partitioner that balances the computational loads and minimizes the runtime interprocessor communication. Several excellent parallel partitioners are now available [4, 5, 10]; however, the results presented in this paper use PMeTiS [6].
- **Processor remapping and data movement:** To map new partitions to processors while minimizing the cost of redistribution, PLUM first constructs a *similarity matrix*. This matrix indicates how the vertex weights of the new subdomains are distributed over the processors. Three general metrics: TotalV, MaxV, and MaxSR, are used to model the remapping cost on most multiprocessor systems [8]. Both optimal and heuristic algorithms for minimizing these metrics are available within PLUM.
- **Cost model metrics:** PLUM performs data remapping in a bulk synchronous fashion to amortize message start-up costs and obtain good cache performance. The procedure is similar to the superstep model of BSP [9]. The expected redistribution cost for a given architecture can be expressed as a linear function of MaxSR. The machine-dependent parameters are determined empirically.

2.1. Helicopter Rotor Test Case

The computational mesh used to evaluate the PLUM load balancer is the one used to simulate an acoustics wind-tunnel experiment of a UH-1H helicopter rotor blade [7]. A cut-out view of the initial tetrahedral mesh is shown in Fig. 1. Three refinement strategies, called Real_1, Real_2, and Real_3, are studied, each subdividing varying fractions of the domain based on an error indicator calculated from the flow solution. This increased the number of mesh elements from 60,968 to 82,489, 201,780, and 321,841, respectively.

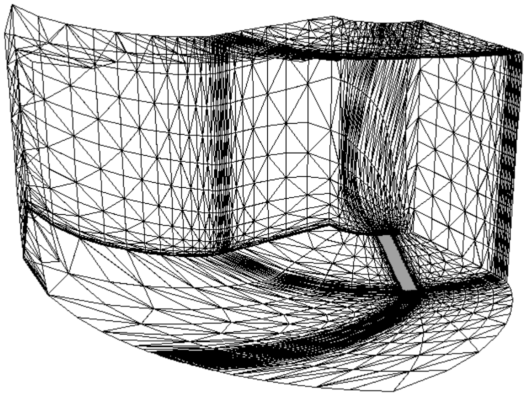

Figure 1. Cut-out view of the initial mesh for the helicopter rotor blade experiment.

2.2. Experimental Results

The three left plots in Fig. 2 illustrate parallel speedup for the three edge-marking strategies on an 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 it returns the biggest benefit by predictively load balancing the refined mesh. The best results are for Real_3 with remapping before refinement, showing an efficiency greater than 87% on 32 processors.

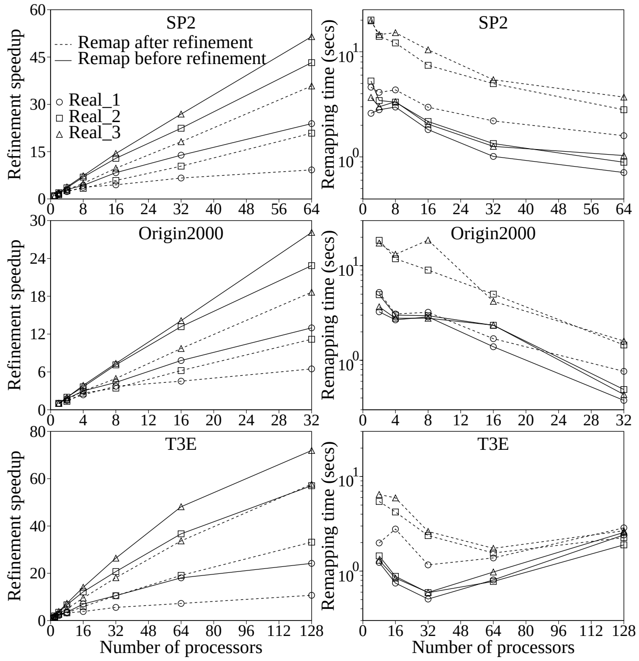

Figure 2. Refinement speedup (left) and remapping time (right) within PLUM on an SP2, Origin2000, and T3E, when data is redistributed after or before mesh refinement.

The three right plots in Fig. 2 show the corresponding remapping times. 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.

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 other two machines. However, for $P = 64$ and 128, the remapping overhead 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 also affected by the interprocessor communication pattern. One solution would be to take advantage of the T3E's ability to efficiently perform one-sided communication.

3. Topology-Independent Load Balancer

In this section, we describe a dynamic load balancer based on a *symmetric broadcast network* (SBN), which takes into account the global view of system loads among the processors. The SBN is a robust, topology-independent communication pattern (logical or physical) among P processors in a multicomputer system [3]. An SBN of dimension $d \geq 0$, denoted as SBN(d), is a $(d+1)$-stage interconnection network with $P = 2^d$ processors in each stage. It is constructed recursively as follows. A single node forms the basis network SBN(0). For $d > 0$, an SBN(d) is obtained from a pair of SBN($d-1$)s by adding a communication stage in the front with the following additional interprocessor connections: (i) node i in stage 0, is made adjacent to node $j = (i + P/2) \bmod P$ of stage 1 and (ii) node j in stage 1 is made adjacent to the node in stage 2 which was the stage 0 successor of node i in SBN($d-1$).

The proposed SBN-based load balancer takes into account a global view of the system and makes it effective for adaptive grid applications. Prior work [2] has demonstrated the viability of this SBN approach. It processes two types of messages: (i) load balance messages when a load imbalance is detected, and (ii) job distribution messages to reallocate jobs. We give a brief description of the various parameters and policies involved in our implementation.

- **Weighted queue length:** This is to take into account all the system variables like computation, communication, and redistribution costs, that affect the processing of a local queue. Note that no redistribution cost is incurred if the data set is available on the local processor. Similarly, the communication cost is zero if the data sets of all adjacent vertices are stored locally.
- **Prioritized vertex selection:** When selecting vertices to be processed, the SBN load balancer takes advantage of the underlying structure of the adaptive grid and defers local execution of boundary vertices as long as possible because they may be migrated for more efficient execution. Thus, vertices with no communication and redistribution costs are executed first.
- **Differential edge cut:** This is the total change in the communication and redistribution costs if a vertex were moved from one processor to another. The vertex with the smallest differential edge cut is chosen for migration. This policy strives to maintain or improve the cut size during the execution of the load balancing algorithm.
- **Data redistribution policy:** Data redistribution is performed in a *lazy* manner, i.e., the non-local data set for a vertex is not moved to a processor until it is about to be executed. Furthermore, the data sets of all adjacent ver-

tices are also migrated at that time. This policy greatly reduces the redistribution and communication costs by avoiding multiple migrations of data sets.

3.1. Unsteady Simulation Test Case

To evaluate the SBN framework, a computational grid is used to simulate an unsteady environment where the adapted region is strongly time-dependent. This experiment is performed by propagating a simulated shock wave through the initial grid shown at the top of Fig. 3. 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. Performance is measured at nine successive adaptation levels during which the mesh increases from 50,000 to 1,833,730 elements.

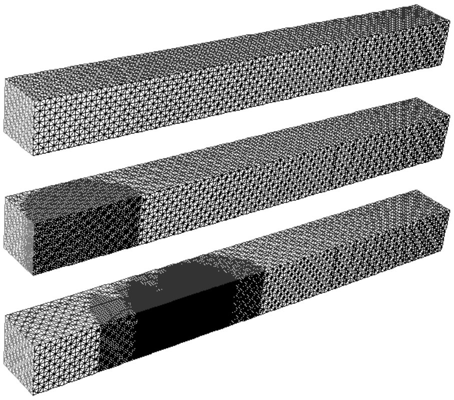

Figure 3. Initial and adapted meshes (after levels 1 and 5) for the simulated unsteady experiment.

3.2. Experimental Results

Table 1 presents performance results on an SP2, averaged over the nine levels of adaptation. In addition to achieving excellent load balance, the redistribution cost (expressed as MaxSR, the maximum number of vertices moved in and out of any processor) is significantly reduced. However, the edge cut percentages are somewhat higher, indicating that the SBN strategy reduces the redistribution cost at the expense of a slightly higher communication cost.

Table 2 gives the number of bytes that were transferred between processors during the load balancing and the job distribution phases. The number of bytes transferred is also expressed as a percentage of the available bandwidth. (A wide-node SP2 has a message bandwidth of 36 megabytes/second and a message latency of 40 micro seconds.) The results demonstrate that the cost of vertex migration is significantly greater than the cost of actually balancing the system load. An extrapolation of the results using exponential curve-fitting indicates that normal speedup will not scale for $P > 128$.

	Edge Cut			Load
P	Before	After	MaxSR	Imbalance
2	2.88%	5.51%	80,037	1.00
4	7.27%	10.76%	76,665	1.00
8	12.71%	16.35%	53,745	1.00
16	19.40%	23.87%	46,825	1.01
32	24.42%	30.41%	28,031	1.02

Table 1. Grid adaptation results on an SP2 using the SBN-based load balancer.

	Balancing Messages		Migration Messages	
P	Volume	Bandwidth	Volume	Bandwidth
2	0.342 MB	0.00%	3.919 MB	3.67%
4	0.150 MB	0.00%	7.939 MB	7.44%
8	0.463 MB	0.01%	25.397 MB	23.79%
16	0.581 MB	0.02%	30.454 MB	28.53%
32	1.550 MB	0.12%	38.244 MB	35.83%

Table 2. Communication overhead of the SBN load balancer.

Table 3 shows the percentage of time spent in the SBN load balancer compared to the execution time required to process the mesh adaptation application. The three columns correspond to three types of load balancing activities: (i) the time needed to handle balance related messages, (ii) the time needed to migrate messages from one processor to another, and (iii) the time needed to select the next vertex to be processed. The results show that processing related to the selection of vertices is the most expensive phase of the SBN load balancer. However, the total time required to load balance is still relatively small compared to the time spent processing the mesh.

We were able to directly port the SBN methodology from an SP2 to an Origin2000 without any code modifications. This demonstrates the architecture independence of our load balancer. The load imbalance factors were almost identical on the two machines. The edge cut values were consistently larger on the Origin2000, but the MaxSR values were larger only when using less than 16 processors.

Some of the differences in performance results on these

P	Balancing Activity	Migration Activity	Vertex Selection
2	0.0053%	0.0014%	0.4530%
4	0.0087%	0.0069%	2.0381%
8	0.1745%	0.0569%	2.8386%
16	0.2669%	0.0629%	0.8845%
32	0.1154%	0.0774%	2.1043%

Table 3. Overhead of the SBN load balancer.

two machines are due to additional refinements that were implemented prior to running the experiments on the Origin2000. First, the algorithm for distributing jobs was modified to guarantee that the processor initiating load balancing always had sufficient workload. This reduces the total number of balancing related messages but increases the number of vertices migrated, especially for small numbers of processors. Second, data sets corresponding to groups of vertices were moved at a time. This bulk migration strategy reduces the volume of migration messages by more than 80% in our experiments.

4. PLUM vs. SBN

Let us highlight some of the differences between the two dynamic load balancers described in this paper.

- Processing is temporarily halted under PLUM while the load is being balanced. The SBN approach, on the other hand, allows processing to continue asynchronously. This feature allows for the possibility of utilizing latency-tolerant techniques to hide the communication overhead of redistribution.
- Under PLUM, suspension of processing and subsequent repartitioning does not guarantee an improvement in the quality of load balance. In contrast, the SBN approach always results in improved load balance.
- PLUM redistributes all necessary data to the appropriate processors immediately before processing. SBN, however, migrates data to a processor only when it is ready to process it, thus reducing the redistribution and communication overhead.
- PLUM, unlike SBN, performs remapping predictively. This results in a significant reduction of data movement during redistribution and improves the overall efficiency of the refinement procedure.
- Load balancing under PLUM occurs *before* the solver phase of the computation, whereas SBN balances the load *during* the solver execution. We therefore cannot directly compare PLUM and SBN-based load balancing, since their relative performance is solver dependent.

5. Summary

We have demonstrated the portability of our novel approaches to solve the load balancing problem for adaptive unstructured grids on three state-of-the-art commercial parallel machines. The experiments were conducted using actual workloads of both steady and unsteady grids obtained from real-life applications. We are currently examining the portability of our software on other platforms (such as shared-memory environments or networks of workstations) as well as employing various parallel programming models.

Acknowledgements

This work is supported by NASA under Contract Numbers NAS 2-14303 with MRJ Technology Solutions and NAS 2-96027 with Universities Space Research Association, and by Texas Advanced Research Program Grant Number TARP-97-003594-013.

References

[1] R. Biswas and L. Oliker. Experiments with repartitioning and load balancing adaptive meshes. Technical Report NAS-97-021, NASA Ames Research Center, 1997.

[2] S. Das, D. Harvey, and R. Biswas. Parallel processing of adaptive meshes with load balancing. In *27th Intl. Conf. on Parallel Processing*, pages 502–509, 1998.

[3] S. Das and S. Prasad. Implementing task ready queues in a multiprocessing environment. In *Intl. Conf. on Parallel Computing*, pages 132–140, 1990.

[4] J. Flaherty, R. Loy, C. Ozturan, M. Shephard, B. Szymanski, J. Teresco, and L. Ziantz. Parallel structures and dynamic load balancing for adaptive finite element computation. *Applied Numerical Mathematics*, 26:241–263, 1998.

[5] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report 95-035, University of Minnesota, 1995.

[6] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report 96-036, University of Minnesota, 1996.

[7] L. Oliker and R. Biswas. PLUM: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52:150–177, 1998.

[8] L. Oliker, R. Biswas, and H. Gabow. Performance analysis and portability of the PLUM load balancing system. In *Euro-Par'98 Parallel Processing*, Springer-Verlag LNCS 1470:307–317, 1998.

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

[10] C. Walshaw, M. Cross, and M. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 47:102–108, 1997.