Parallel Dynamic Load Balancing Strategies for Adaptive Irregular Applications

Rupak Biswas

Computer Sciences Corporation, MS T27A-1, NASA Ames Research Center, Moffett Field, CA 94035. E-mail: `rbiswas@nas.nasa.gov`

Sajal K. Das, Daniel J. Harvey

Dept. of Computer Science & Engineering, The University of Texas at Arlington, Arlington, TX 76019. E-mail: `{das,harvey}@cs.unt.edu`

Leonid Oliker

NERSC, MS 50F, Lawrence Berkeley National Laboratory, Berkeley, CA 94720. E-mail: `loliker@lbl.gov`

Abstract

Dynamic unstructured mesh adaptation is a powerful technique for solving computational problems with evolving physical features; however, an efficient parallel implementation is rather difficult because of the load imbalance that mesh adaptation creates. To address this problem, we have developed two dynamic load balancing strategies for parallel adaptive irregular applications. The first, called PLUM, is an architecture-independent framework particularly geared toward adaptive numerical computations and requires that all data be globally redistributed after each adaptation to achieve load balance. The second is a more general-purpose topology-independent 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. Results indicate that both PLUM and the SBN-based approach have their relative merits, and that they achieve excellent load balance at the cost of minimal extra overhead.

Key words: Unstructured mesh adaptation, global dynamic load balancing, symmetric broadcast networks, parallel computing

1 Introduction

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 because of the load imbalance that mesh adaptation creates. Dynamic load balancing aims to balance processor workloads at runtime while attempting to minimize the communication between processors. A problem is therefore load balanced when processors have nearly equal loads with reduced communication among themselves. The ultimate objective, of course, is to assign work to processors so that the total simulation runtime is minimized. With the proliferation of parallel computing, dynamic load balancing has also become extremely important in areas other than scientific computing like task scheduling, discrete event simulation, and web server applications.

Standard fixed-mesh numerical methods to solve a full-scale realistic problem can be made more cost-effective by locally refining and coarsening the computational mesh to capture physical phenomena of interest. However, when executed on a parallel machine, mesh adaptation could cause load imbalance among the processors. This, in turn, would lead to idle processors and adversely affect the total execution time. It is therefore imperative to have an efficient dynamic load balancing mechanism as part of the overall solution scheme. In addition, since the computational mesh will be frequently adapted for unsteady flows, the runtime load may also need to be frequently rebalanced (in the worst case, at each adaptation step). This implies that the dynamic load balancing procedure itself must not pose a major overhead. Although several dynamic load balancers have been proposed for multiprocessor platforms [1–6], most of them are inadequate for adaptive unstructured grid applications. This motivates our work.

Recently, 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 computational mesh is globally repartitioned from scratch after each adaptation, but a smart remapping technique is used to reassign the new partitions among the processors in a way that minimizes the data movement overhead. This redistribution cost can be estimated using a variety of metrics. The new partitioning is accepted if the migration cost is offset by the gain resulting from a better load balance. A brief description of PLUM is given in Section 3.

We also describe another new dynamic load balancing strategy that is based on utilizing a robust communication pattern among processors, called symmetric broadcast networks (SBN), originally proposed in [8]. Our earlier experiments with synthetic loads [2] have demonstrated that an SBN-based load balancer achieves superior performance when compared to other popular techniques [9–12]. The SBN algorithm is adaptive and decentralized in nature, and can be ported to any topological architecture through efficient embedding techniques. An overview of this approach is given in Section 4.

Both PLUM and the SBN-based load balancing strategies have been implemented on IBM SP2 and SGI Origin2000 machines, and their performance analyzed for an adaptive unsteady workload which is generated by propagating a simulated shock wave through a cylindrical volume. Results, presented in Section 5, show that the SBN approach reduces the data remapping cost at the expense of a higher interprocessor communication overhead. In dynamic applications where the data redistribution cost dominates the processing and communication costs, this is an acceptable trade-off.

2 Dynamic Load Balancing

Dynamic load balancing for unstructured grids consists of two main components: partitioning the computational mesh (to balance processor workloads and minimize runtime interprocessor communication), and mapping the partitions to processors (to minimize data movement). Diffusive partitioners modify existing partitions, while global partitioners generate new partitions from scratch that need to be remapped to processors.

2.1 Partitioning

Graph partitioning is NP-complete; thus, research has been focused on developing efficient heuristic algorithms. Several graph partitioners have been developed over the years, particularly for static grids. Significant progress has been made in improving the partitioning heuristics as well as in generating high-quality software.

The most general approach to graph partitioning is to use generic combinatorial optimization techniques based on cost functions. Simulated annealing [13] and genetic algorithms [14] are two such techniques but require properly setting a large number of parameters, making it difficult to obtain successful partitionings efficiently. Clustering is another intuitive approach. The greedy strategy described in [15] and bandwidth reduction algorithms like Reverse Cuthill-McKee (RCM) [16] belong to this class of partitioners. The bad aspect ratio problem associated with RCM can be somewhat reduced if the scheme is used recursively, as in recursive graph bisection (RGB) [17].

Geometry-based algorithms partition the graph by exploiting its geometric properties. Recursive coordinate bisection (RCB) [17] uses spatial geometric coordinates, whereas recursive inertial bisection (RIB) [18] is based on inertial coordinates. A completely different class of algorithms is based on spectral methods. Recursive spectral bisection (RSB) [17] is the most famous of such partitioners. Multidimensional spectral partitioning [19] improves RSB runtimes by performing a k-way partitioning at each recursive step. Many of these geometric and spectral methods are used with the Kernighan-Lin refinement strategy [20] to improve the partitioning quality.

However, the most successful state-of-the-art partitioners use multilevel algorithms [21–23] that reduce partitioning times considerably by contracting the graph, partitioning the coarsened graph, and then refining it to obtain a partition for the original graph. ParMETIS [22] and JOSTLE [23] are currently the fastest multilevel schemes, are available in various flavors, and can be run either as partitioners from scratch or as diffusive repartitioners.

2.2 Remapping

The overall effectiveness of repartitioning algorithms is determined by how successful they are in load balancing the computations while minimizing the edge-cut, as well as reducing the cost associated with redistributing the load in order to realize the new partitioning. Data redistribution is generally considered the most expensive phase in dynamic load balancing. The migration of mesh objects incurs a number of costs such as the communication overhead of remote-memory latency, and the computational overhead of rebuilding internal and shared data structures. Since global partitioners do not consider the redistribution cost when generating subdomains, an intelligent algorithm is needed to map the new partitions onto the processors such that the data migration cost is minimized. Diffusive repartitioners explicitly attempt to minimize this data redistribution cost when generating new subdomains. These strategies are generally successful when there are gradual changes in the load distribution. However, experimental results [24,25] have indicated that diffusive schemes may not be well suited for problems which incur dramatic shifts in processor workloads between redistributions, such as unsteady adaptive applications.

Diffusion was first presented as a method for load balancing in [1]. The process can be mapped onto the diffusion equation, and much is known about its properties. In particular, it can be shown that this process will eventually converge. A nonlinear variant of this scheme which considers strip decompositions of the domain was presented in [5] and shown to have better convergence characteristics than the standard diffusion method.

Tiling is another approach [26] where each processor is considered a neighborhood center, a neighbor being any processor with which the given processor shares its subdomain boundary. Processors within a neighborhood are balanced with respect to one another using local performance measurements. Iterative tree balancing (ITB) [27] follows the basic tiling strategy; however, the algorithm views workload requests as forming a forest of trees rather than considering a neighborhood of processors. ITB incorporates a more global information, and has an improved worst case load imbalance if enough iterations are permitted.

Local iterative techniques are sometimes unsuitable for dynamically balancing unsteady numerical calculations. Such applications are prone to shifting the load distribution from one adaptation phase to the next, causing small regions of the domain to suddenly incur high computational costs. Local diffusion techniques would require several iterations before global convergence, or accept an unbalanced load in exchange for faster performance. Also, by limiting load movement to nearest neighbors, several hops may be needed for a work unit to arrive at its final destination. Since the remapping must be frequently applied, its cost can become a significant part of the overall performance and must therefore be minimized. By moving large chunks of work units directly to their destinations, the high start-up cost of interprocessor communication can be amortized. This is the strategy that has been implemented within the PLUM and SBN frameworks.

3 Architecture-Independent PLUM Load Balancer

PLUM is an automatic and portable load balancing environment, specifically created to handle adaptive unstructured grid applications. It differs from most other similar load balancers in that it dynamically balances processor workloads with a global view. Prior work [7,24,28] has successfully demonstrated its viability on large numbers of processors, its portability across computer platforms, and its effectiveness for various test cases involving adaptive unstructured grids. In this paper, we compare its performance with the SBN-based approach (described in Section 4).

Figure 1 provides an overview of PLUM. After an initial partitioning and mapping of the unstructured grid, a solver executes several iterations of the application. A mesh adaptation procedure is invoked when the computational mesh needs 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 necessary. 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 expected gain

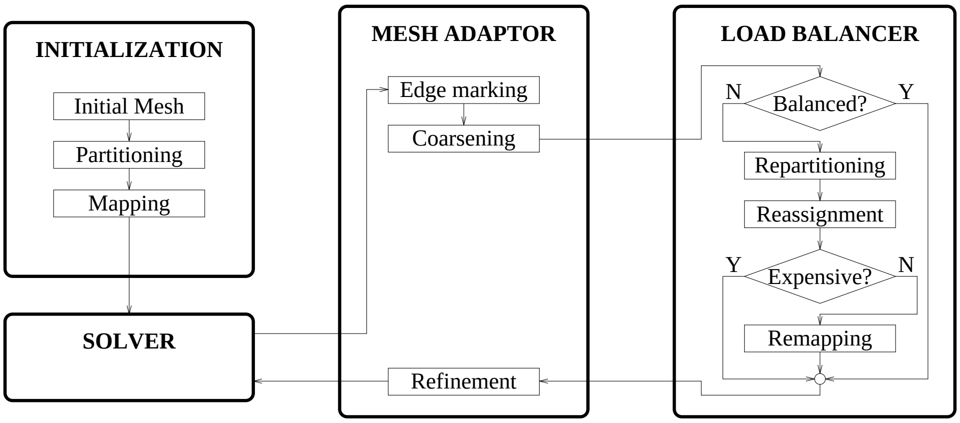

Fig. 1. Overview of PLUM load balancer.

resulting from a better load balance exceeds the estimated remapping cost, the grid is remapped among the processors before resuming the computation. A brief description of the salient features of PLUM is given in the following subsections.

3.1 Reusing the Initial Dual Graph

PLUM always uses the dual of the initial mesh for load balancing, thus keeping 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. The weight, W_v, of a dual graph vertex v models the computational workload, and is set to the number of leaf elements in the corresponding refinement tree. This is because only those elements with no children participate in the actual computation. The weight, R_v, of v models the redistribution cost, and is the total number of elements in the refinement tree. This is because all descendents of the root element must be moved from one partition to another when the load is to be rebalanced. Lastly, the weight, C_e, of a dual graph edge e models the communication cost, and is set to the number of corresponding faces in the computational mesh. These three weights are used to balance the workload among processors, minimize the runtime communication, and optimize the data movement cost after a repartitioning.

3.2 Parallel Repartitioning

After a mesh adaptation, PLUM usually needs to rebalance the processor workloads and minimize the interprocessor communication. PLUM can use any general-purpose partitioner; however, for it to be viable, the repartitioning must be performed rapidly. As discussed in Section 2.1, several excellent parallel partitioners are now available. For the results presented in this paper, the ParMETIS parallel multilevel partitioner was used [22]. It reduces the size of the graph by collapsing vertices and edges, partitions the smaller problem, and uncoarsens the graph back to the original size. ParMETIS uses a greedy graph bisection algorithm for partitioning the coarsest graph, and uncoarsens it by using a combination of boundary greedy and Kernighan-Lin [20] refinement. A detailed performance comparison of ParMETIS with other partitioners is given in [24].

3.3 Processor Remapping

The goal of processor reassignment is to find a mapping between partitions and processors that minimizes the data redistribution cost. In theory, the number of new subdomains can be an integer multiple F of the number of processors. Each processor is then assigned F unique subdomains. The rationale for allowing multiple subdomains per processor is that remapping at a finer granularity reduces the volume of data movement at the cost of a slightly larger partitioning time. We first compute a similarity measure that indicates how the new subdomains are distributed over the P processors. The measure is represented as a matrix S, where entry S_{ij} is the sum of the R_v values of all the dual graph vertices in new partition j that already reside on processor i. Figure 2 shows an example of a similarity matrix for four processors where two partitions are assigned to each processor. Only the non-zero entries are shown for clarity.

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. A similarity matrix after processor reassignment.

Various cost functions are usually needed to solve the processor reassignment problem using S for different machine architectures. We have developed three general metrics: `TotalV`, which minimizes the total volume of data moved among all the processors; `MaxV`, which minimizes the maximum flow of data to *or* from any single processor; and `MaxSR`, which minimizes the sum of the maximum flow of data to *and* from any processor. Experimental results [7,28] have indicated the usefulness of these metrics in predicting the actual remapping cost.

The `TotalV` metric assumes that remapping time will be reduced by decreasing network contention and the total number of elements moved. Finding the optimal mapping for `TotalV` can be reduced to solving the *maximally weighted bipartite graph* (MWBG) problem. The optimal algorithm has been implemented with a runtime of $O(|V|^3)$ [28], where $|V|$ represents the number of partitions. 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. We can solve for the `MaxV` metric 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. The optimal BMCM algorithm has been implemented with a runtime of $O(|V|^{1/2}|E|\log|V|)$ [28], where $|E|$ is the number of entries in the similarity matrix ($|E| \approx |V|^2$). 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. This is referred to as the *double* bottleneck maximum cardinality matching (DBMCM) problem. We have developed an algorithm for computing the minimum DBMCM with a runtime of $O(|V|^{1/2}|E|^2\log|V|)$ [28].

A greedy heuristic algorithm for solving the reassignment problem in $O(|E|)$ steps has also been developed, and shown to generate a solution that can never result in a data movement cost that is more than twice that of the optimal `TotalV` assignment [7]. A detailed comparison of reassignment algorithms using different metrics is given in [28].

3.4 Remapping Cost Model

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. However, a model is needed to quickly predict the expected redistribution cost for a given architecture.

The expected time for the redistribution procedure can be expressed as $\gamma \times$

$\texttt{MaxSR} + O$, where $\texttt{MaxSR} = \max(\texttt{ElemsSent}) + \max(\texttt{ElemsRecd})$, γ represents the total computation and communication cost to process each redistributed element, and O is the predicted sum of all constant overheads [7]. This model demonstrates the need to use the `MaxSR` metric for processor reassignment because the computational overhead of our remapping algorithm can be reduced by minimizing `MaxSR`. Since the computational workload is architecture independent, we are effectively solving two load balancing problems separated by a collective communication. Moreover, by reducing `MaxSR`, we can achieve a savings in the communication overhead on many bandwidth-rich systems. The values of γ and O can be obtained for each architecture by simple experiments [7].

4 Topology-Independent SBN-Based Load Balancer

Our second load balancer is 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 the processors of a multicomputer system [8]. Before utilizing it to load balance adaptive grids, let us give a brief overview of SBN.

4.1 Symmetric Broadcast Networks

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 processor forms the basis network SBN(0) consisting of a single stage, denoted as stage 0.
- For $d > 0$, an SBN(d) is obtained from a pair of SBN($d-1$)s as follows: (i) The processor labels in the first SBN($d-1$) remain unchanged, but the processors of the second SBN($d-1$) are relabeled from 2^{d-1} to $2^d - 1$;
(ii) Increment the identifiers of the existing stages by one, and create a new communication stage 0 containing processors 0 through $2^d - 1$;
(iii) Connect processor i in stage 0 to processor $j = (i + P/2) \bmod P$ of stage 1, and processor j in stage 1 to the processor in stage 2 (if present) which was the stage 0 successor of processor i in SBN($d-1$).

Figure 3 depicts an SBN(2), recursively constructed from two SBN(1)s. Note that an SBN(d) defines unique communication patterns (or broadcast trees) among the processors in the network. Precisely, for any source processor p at stage 0, where $0 \leq p < P$, there exists a unique broadcast tree T_p of height

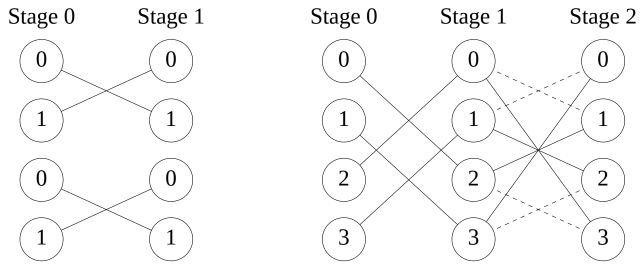

Fig. 3. Construction of SBN(2) from a pair of SBN(1)s. The new connections are shown by solid lines and the original connections by dashed lines.

$d = \log P$ such that each of the 2^d processors appears exactly once.

Furthermore, the SBN communication pattern for any source processor p can be derived from the template tree with processor 0 as the source. For example, let N_p^s be a processor at stage s in the broadcast tree T_p. Then $N_p^s = N_0^s \oplus p$, where $\oplus$ is the exclusive-OR operator, thus leading to $T_p = T_0 \oplus p$. The predecessor and successors of each processor are also uniquely defined by specifying the source and the communication stage. As shown in [29], the versatility of the SBN lies in its efficient embeddings into other topologies such as hypercubes and meshes.

4.2 Load Balancing

The proposed SBN-based load balancer can be classified as adaptive, decentralized, and global, making it effective for adaptive grid applications. The SBN load balancer processes two types of messages: balance messages and distribution messages.

A balance message is broadcast when a processor p determines that its weighted queue length `QWgt`(p) is less than a minimum threshold, `MinTh`. A balance message is also broadcast if `QWgt`(p) is greater than a maximum threshold, `MaxTh`, or if distributing the excess workload will result in other processors exceeding `MaxTh`. As the balance message is propagated through the SBN, the global weighted queue length (`GWLen`) and the weighted system load level (`WSysLL`) are computed. Distribution messages are used to migrate work when `QWgt`(p) is greater than `MaxTh`. After `WSysLL` is calculated, a distribution message is broadcast through the SBN in order to route work to the lightly-loaded processors and to update the threshold values (`MinTh` and `MaxTh`). As a result, all the processor workloads are balanced.

We now discuss the various parameters and other details involved in the SBN-based load balancer implementation, a preliminary version of which appeared in [30].

- Weighted Queue Length:
  The queue length (computation time) of a processor p is not an accurate estimate of the total time required to complete its work, particularly in applications where the grid is adapted. We therefore define a metric called weighted queue length, $\texttt{QWgt}(p)$, that also considers the communication and redistribution costs. If W_v is the computational cost to process a dual graph vertex v, C_v^p is the communication cost to interact with the vertices adjacent to v but whose data sets are not local to p, and R_v^p is the redistribution cost to copy the data set for v to p from another processor (see Section 3.1), then

$$\texttt{QWgt}(p) = \sum_{v \text{ assigned to } p} (W_v + C_v^p + R_v^p).$$

  Note that if the data set for v is already assigned to p, no redistribution cost is incurred ($R_v^p = 0$). Similarly, if the data sets of all the vertices adjacent to v are also already assigned to p, the communication cost, $C_v^p = 0$.
- Weighted System Load:
  The weighted system load is defined as:

$$\texttt{WSysLL} = \left\lceil \frac{1}{P} \sum_{p=1}^{P} \texttt{QWgt}(p) \right\rceil,$$

  where P is the total number of processors used. Assuming that the workload is perfectly balanced among the processors, $\texttt{WSysLL}$ estimates the time required to process the grid and reflects the computation, communication, and redistribution costs in the current grid-to-processor assignment. Hence, a global view of the system is captured.
- Prioritized Vertex Selection:
  When selecting dual graph vertices to process, the SBN load balancer utilizes the underlying grid structure to defer execution of boundary vertices as long as possible because they may be migrated for more efficient execution. Thus, the vertex to be processed next is selected such that it minimizes the overall edge cut of the adapted grid. To enable this, a priority min-queue is maintained, where the priority of vertex v in processor p is given by $(C_v^p + R_v^p)/W_v$. Therefore, vertices with no communication and redistribution costs are executed first, while those with high communication or redistribution overhead relative to their computational weight are processed last. Basically, internal vertices are processed before those on partition boundaries.
- Differential Edge Cut:
  To balance processor workloads, an optimal policy for vertex migration is required. Assume that processor p needs to reassign some of its vertices to another processor q. The SBN load balancer running on p calculates the

differential edge cut [1], $\Delta\texttt{Cut}(v)$, for each vertex v that is queued locally as follows:

$$\Delta\texttt{Cut}(v) = R_v^q - R_v^p + C_v^q - C_v^p.$$

A negative value indicates a reduction in communication and redistribution costs if v is migrated from p to q, hence favoring the migration of vertices with the largest absolute reduction in these costs. The vertex w with the smallest value of $\Delta\texttt{Cut}(w)$ is chosen for migration. The vertices adjacent to w that are queued for processing on p are also targeted. The entire process is repeated if more work needs to be transferred from p to q. This migration policy therefore strives to maintain or improve the cut size during the execution of the load balancing algorithm.

- Data Redistribution Policy:
  Data is redistributed in a lazy manner, i.e. the data set for a vertex v in processor p is not moved to processor q until q is ready to execute v (q notifies p when this happens). Moreover, the data sets of all vertices adjacent to v that are assigned to q are transferred as well. This policy significantly reduces the redistribution and communication costs by avoiding multiple migration of data sets and having resident on q all adjacent vertices of v while q processes v. In other words, the communication overhead is reduced by considering the underlying grid structure.

4.3 Differences with PLUM

The SBN-based load balancer differs from PLUM in several important respects that are itemized below:

- Load balancing under PLUM requires the temporary suspension of useful processing in order to generate new partitions and redistribute the data. Instead, the SBN-based approach allows processing to continue while the load is dynamically balanced. This feature makes it possible to use latency-tolerant techniques to hide the communication and redistribution costs during processing.
- Under PLUM, the suspension of processing and subsequent repartitioning does not guarantee an improvement in load balance quality. This is because the adapted grid is not redistributed if the remapping cost exceeds the benefits of load balancing. In contrast, the SBN approach will always improve the load balance among the processors.
- PLUM redistributes all necessary data to the appropriate processors before the computations are resumed. This predictive remapping improves

[1] Here we are deviating from the usual definition of edge cut to account for the dynamic nature of the SBN-based load balancer.

the overall efficiency of the grid refinement procedure. SBN, however, reduces redistribution and communication costs by migrating remote data to a processor only when it is ready to process the data.

5 Experimental Results

All experiments reported in this paper were performed on the SP2 and the Origin2000 machines at NASA Ames Research Center. The IBM 9076 Scalable POWERParallel SP2 is a distributed-memory system where the nodes, each consisting of a six-instruction issue superscalar RISC6000 processor, are connected through a high-performance switch called the Vulcan chip. The topology of the switch is an any-to-any packet switched network similar to an Omega network. The SGI Origin2000, on the other hand, has a distributed shared-memory architecture and is the first commercially available 64-bit cache-coherent nonuniform memory access (CC-NUMA) system. A small high-performance switch connects two CPUs (which are superscalar RISC processors), memory, and I/O. This module, called a node, is then connected to other nodes in a hypercube fashion.

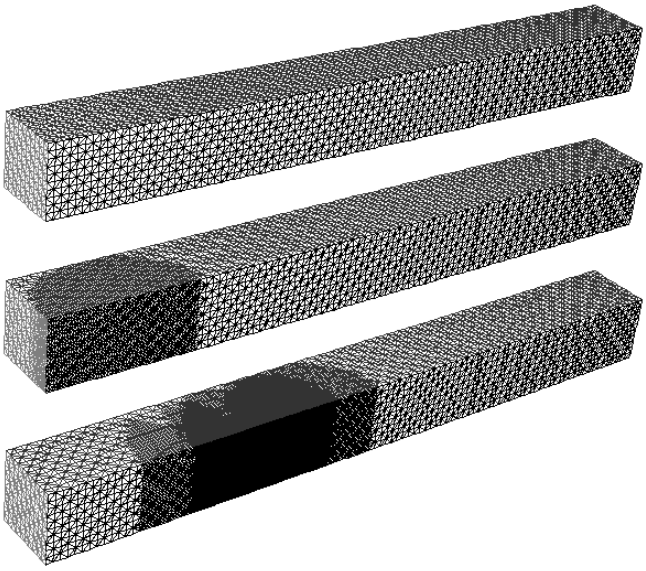

Fig. 4. Initial and adapted meshes (after levels 1 and 5) for the simulated unsteady experiment.

To study the effectiveness of our load balancers for realistic applications, 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. 4. 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 size of the computational mesh increased from 50,000 elements to 1,833,730.

The following metrics were chosen to compare the performance of PLUM and the SBN-based load balancers. Recall that v denotes a vertex to be processed and P is the total number of processors.

- Load Imbalance Factor:
  This metric is the ratio of the workload on the most heavily-loaded processor to the average load across all processors, and should be as close to unity as possible. For PLUM, the workload on processor p is the sum of the W_v weights for all dual graph vertices v assigned to p. For the SBN-based load balancer, this metric is formulated as

$$\texttt{LoadImb} = \max_{p \in P} \texttt{QWgt}(p) \;/\; \texttt{WSysLL}.$$

- Cut Percentage:
  This metric represents the runtime interaction between adjacent dual graph vertices residing on different processors, and should be as small as possible. For PLUM, it is the weighted percentage of all the edges that are cut by the ParMETIS partitioner. In the SBN approach, it is calculated as

$$\texttt{Cut\%} = 100 \times \sum_{p \in P} \sum_{v \text{ assigned to } p} C_v^p \Big/ \sum_{e \text{ in mesh}} C_e \,,$$

  where C_e is the weight of edge e in the initial dual graph.
- Maximum Redistribution Cost:
  Since a processor can either be sending or receiving data, the overhead for the two phases is modeled as a sum of two costs in this metric:

$$\texttt{MaxSR} = \max_{p \in P} \Big\{ \sum_{v \text{ sent from } p} R_v^p \Big\} + \max_{p \in P} \Big\{ \sum_{v \text{ recv by } p} R_v^p \Big\}.$$

  Recall from Section 3.4 that minimizing `MaxSR` guarantees a reduction in the total data redistribution overhead.

Both the PLUM and the SBN-based load balancers were implemented using the MPI message-passing library. Performance results, averaged over nine levels of adaptation, are presented in Table 1. These values were obtained on the Origin2000, and are platform independent. Both strategies achieve excellent load balance quality (`LoadImb`); however, the SBN-based algorithm significantly reduces the redistribution cost (`MaxSR`) from that obtained with PLUM, more so for the larger numbers of processors. Compared to PLUM, the SBN algorithm also incurs a smaller overhead in terms of the total communication

Table 1
Performance results of the PLUM and the SBN-based load balancers

	`LoadImb`		`Cut%`		`MaxSR`		Comm. Vol.	
P	PLUM	SBN	PLUM	SBN	PLUM	SBN	PLUM	SBN
2	1.00	1.01	3.29	7.65	157,542	158,085	7.625	3.921
4	1.00	1.00	4.01	12.81	143,227	114,652	14.264	7.250
8	1.01	1.01	5.77	18.33	129,859	65,824	26.092	18.356
16	1.02	1.01	7.81	29.25	103,090	35,389	36.787	27.658
32	1.04	1.02	10.94	36.47	63,270	19,446	41.113	35.268

volume (in MBytes), which is proportional to the `TotalV` metric. (Observe that the total communication volume increases with the number of processors, while the `MaxSR` metric decreases.) On the other hand, the ParMETIS partitioner used in PLUM generates dramatically smaller edge cuts (`Cut%`), thereby minimizing the runtime interprocessor communication. Thus there is a trade-off here: SBN reduces the data redistribution overhead at the expense of greater runtime communication. This is due to the lazy approach of data migration used by the SBN-based load balancer, but may be an acceptable compromise for applications where the data migration cost is dominant.

Table 2 shows the fraction of time spent in the load balancer compared to the total time required to process the mesh adaptation application, for both PLUM and SBN. Notice that the partitioning overhead of PLUM is dramatically less than the balancing cost of SBN. This is due to the state-of-the-art ParMETIS partitioner, which uses the dual graph information to efficiently compute a new mesh distribution. The SBN balancer, on the other hand, is dominated by the cost of vertex selection. This algorithm dynamically chooses

Table 2
Percentage overhead of the two load balancers

	PLUM				SBN			
	Partitioning		Remapping		Balancing		Distributing	
P	SP2	O2000	SP2	O2000	SP2	O2000	SP2	O2000
2	0.13	0.14	1.45	1.11	0.86	0.75	0.04	0.03
4	0.15	0.15	2.56	2.01	1.15	0.99	0.11	0.08
8	0.17	0.18	3.52	2.83	2.11	1.48	0.20	0.11
16	0.32	0.34	4.04	3.31	2.75	1.89	0.28	0.16
32	0.46	0.50	4.40	3.67	3.01	2.52	0.38	0.20

the next dual graph vertex to be processed, depending on specific runtime criteria. Thus, vertex selection is not as efficient as parallel partitioning, since the balancing data are not available a priori as in the PLUM framework. However, the cost of remapping within PLUM is significantly more expensive than SBN distribution. During PLUM remapping, useful processing is suspended while the mesh is redistributed and data objects are appropriately rebuilt. The SBN approach allows processing to continue while the load is dynamically balanced. This allows the communication overhead of distribution to be overlapped with the processing of the mesh application, substantially reducing the data movement cost compared to traditional remapping schemes. Finally, observe that the Origin2000 is responsible for a smaller overall percentage of remapping/distribution compared to the SP2, for both balancing strategies. This is due the superior network performance of the Origin2000.

6 Conclusions

In this paper, we have described two novel approaches (PLUM and SBN-based) to solving the dynamic load balancing problem for parallel adaptive unstructured grids. We have demonstrated their effectiveness and performance on a computational grid that was used to simulate an unsteady environment where the adapted region is strongly time-dependent. Results indicated that both strategies achieve excellent load balance at the cost of minimal extra overhead (less than 5% for our application). Compared to PLUM, the SBN approach reduces the data remapping cost at the expense of higher interprocessor communication. In dynamic applications where the data redistribution cost dominates the processing and communication costs, this may be an acceptable trade-off. We are currently examining the portability of our software to shared-memory environments and workstation clusters. We also believe that the performance of our load balancers can be improved by exploiting one-sided communication. This will be the subject of future research.

Acknowledgements

The work of the first author was supported by NASA under contract number NAS 2-14303 while he was with MRJ Technology Solutions. The work of the second and third authors was supported by Texas Advanced Research Program grant number TARP-97-003594-013. The work of the fourth author was supported by The Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098.

References

[1] Cybenko, G., Dynamic load balancing for distributed-memory multiprocessors. J. Parallel Distrib. Comput. 7 (1989) 279–301.

[2] Das, S.K., Harvey, D.J., Biswas, R., Adaptive load balancing algorithms using symmetric broadcast networks: Performance study on an IBM SP2. Proc. 26th Intl. Conf. on Parallel Processing, Bloomingdale, IL (1997) 360–367.

[3] Ghosh, B., Muthukrishnan, S., Dynamic load balancing in parallel and distributed networks by random matchings. Proc. 6th ACM Symp. on Parallel Algorithms and Architectures, Cape May, NJ (1994) 226–235.

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

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

[6] Mahapatra, N., Dutt, S., Random seeking: A general, efficient, and informed randomized scheme for dynamic load balancing. Proc. 10th Intl. Parallel Processing Symp., Honolulu, HI (1996) 881–885.

[7] Oliker, L., Biswas, R., PLUM: Parallel load balancing for adaptive unstructured meshes. J. Parallel Distrib. Comput. 52 (1998) 150–177.

[8] Das, S.K., Prasad, S.K., Implementing task ready queues in a multiprocessing environment. Proc. Intl. Conf. on Parallel Computing, Pune, India (1990) 132–140.

[9] Eager, D.L., Lazowska, E.D., Zahorjan, J., A comparison of receiver-initiated and sender-initiated adaptive load sharing. Performance Evaluation 6 (1986) 53–68.

[10] Eager, D.L., Lazowska, E.D., Zahorjan, J., Adaptive load sharing in homogeneous distributed systems. IEEE Trans. on Soft. Engrg. 12 (1986) 662–675.

[11] Lin, F.C.H., Keller, R.M., The gradient model load balancing method. IEEE Trans. on Soft. Engrg. 13 (1987) 32–38.

[12] Shu, W., Wu, M.-Y., Runtime incremental parallel scheduling (RIPS) on distributed memory computers. IEEE Trans. on Parallel Distrib. Sys. 7 (1996) 637–649.

[13] Kirkpatrick, S., Gelatt Jr., C.D., Vechhi, M.P., Optimization by simulated annealing. Science 220 (1983) 671–680.

[14] Khuri, S., Baterekh, A., Genetic algorithms and discrete optimization. Methods of Operations Research 64 (1991) 133–142.

[15] Farhat, C., A simple and efficient automatic FEM domain decomposer. Computers and Structures 28 (1988) 579–602.

[16] Chan, W., George, A., A linear time implementation of the reverse Cuthill-McKee algorithm. BIT 20 (1980) 8–14.

[17] Simon, H.D., Partitioning of unstructured problems for parallel processing. Comput. Sys. in Engrg. 2 (1991) 135–148.

[18] Nour-Omid, B., Raefsky, A., Lyzenga, G.A., Solving finite element equations on concurrent processors. Proc. Symp. on Parallel Computations and their Impact on Mechanics, Boston, MA (1987) 209–218.

[19] Hendrickson, B., Leland, R., Multidimensional spectral load balancing. Technical Report SAND93-0074, Sandia Natl. Labs., Albuquerque, NM (1993).

[20] Kernighan, B.W., Lin, S., An efficient heuristic procedure for partitioning graphs. Bell Systems Technical Journal 49 (1970) 291–307.

[21] Hendrickson, B., Leland, R., A multilevel algorithm for partitioning graphs. Proc. Supercomputing '95, San Diego, CA (1995).

[22] Karypis, G., Kumar, V., Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report 96-036, Dept. of Computer Science, Univ. of Minnesota, Minneapolis, MN (1996).

[23] Walshaw, C., Cross, M., Everett, M.G., Parallel dynamic graph partitioning for adaptive unstructured meshes. J. Parallel Distrib. Comput. 47 (1997) 102–108.

[24] Biswas, R., Oliker, L., Experiments with repartitioning and load balancing adaptive meshes. Grid Generation and Adaptive Algorithms, IMA Volumes in Mathematics and its Applications 113 (1999) 89–111.

[25] Schloegel, K., Karypis, G., Kumar, V., Biswas, R., Oliker, L., A performance study of diffusive vs. remapped load-balancing schemes. Proc. 11th Intl. Conf. on Parallel and Distributed Computing Systems, Chicago, IL (1998) 59–66.

[26] Leiss, E., Reddy, H., Distributed load balancing: Design and performance analysis. W.M. Keck Research Computation Laboratory 5 (1989) 205–270.

[27] Flaherty, J.E., Loy, R.M., Ozturan, C., Shephard, M.S., Szymanski, B.K., Teresco, J.D., Ziantz, L.H., Parallel structures and dynamic load balancing for adaptive finite element computation. Appl. Numer. Math. 26 (1998) 241–263.

[28] Oliker, L., Biswas, R., Gabow, H.N., Parallel tetrahedral mesh adaptation with dynamic load balancing. Parallel Comput. (2000) to appear.

[29] Das, S.K., Harvey, D.J., Performance analysis of an adaptive symmetric broadcast load balancing algorithm on the hypercube. Technical Report CRPDC-95-1, Dept. of Computer Science, Univ. of North Texas, Denton, TX (1995).

[30] Das, S.K., Harvey, D.J., Biswas, R., Parallel processing of adaptive meshes with load balancing. Proc. 27th Intl. Conf. on Parallel Processing, Minneapolis, MN (1998) 502–509.