Parallel Processing of Adaptive Meshes with Load Balancing

Sajal K. Das and Daniel J. Harvey
Dept. of Computer Science & Engineering
The University of Texas at Arlington
Arlington, TX 76019
{das, harvey}@cse.uta.edu

Rupak Biswas
NASA Ames Research Center
Mail Stop T27A-1
Moffett Field, CA 94035
rbiswas@nas.nasa.gov

Abstract

Many scientific applications involve grids that lack a uniform underlying structure. These applications are often also dynamic in nature in that the grid structure significantly changes between successive phases of execution. In parallel computing environments, mesh adaptation of unstructured grids through selective refinement/coarsening has proven to be an effective approach. However, achieving load balance while minimizing interprocessor communication and redistribution costs is a difficult problem. Traditional dynamic load balancers are mostly inadequate because they lack a global view of system loads across processors. In this paper, we propose a novel and general-purpose load balancer that utilizes symmetric broadcast networks (SBN) as the underlying communication topology, and compare its performance with a successful global load balancing environment, called PLUM, specifically created to handle adaptive unstructured applications. Our experimental results on an IBM SP2 demonstrate that the SBN-based load balancer achieves lower redistribution costs than that under PLUM by overlapping processing and data migration.

Key words: Dynamic load balancing, experimental study, IBM SP2, job migration and redistribution, symmetric broadcast networks, unstructured mesh adaptation

1 Introduction

Mesh partitioning is a common approach to parallelize many scientific applications which are generally modeled discretely using a mesh (or grid) of vertices and edges. For maximum efficiency, the computational workloads on the processors have to be balanced and the number of edges that are cut (and hence the overall interprocessor communication cost at runtime) needs to be minimized. For this purpose, each vertex is usually assigned a weight that indicates the amount of computation required to process it. Similarly, each edge in the mesh has an associated weight indicating the amount of interaction between adjacent vertices. To achieve load balance dynamically, portions of the mesh have to be migrated among processors during the course of a computation. Thus, in a multiprocessing environment, the vertex weight contains an additional component that models the cost of redistributing the vertex from one processor to another. These weights are used to minimize the data redistribution cost during the remapping phase.

With adaptive meshes, the grid topology changes during the course of a computation. Traditionally, this class of problems is processed by load balancing the mesh after each adaptation. A number of partitioners designed for this purpose has been proposed in the literature [8, 11, 14, 17, 21]. A majority of the successful partitioners are based on a multilevel approach that has proven to be extremely effective in producing good partitions at reasonable cost. In a multilevel scheme, the grid is first contracted to a small number of vertices and edges, the coarsened grid is next partitioned, and is then finally refined to the original using the Kernighan-Lin replacement algorithm [12]. However, other partitioning methods have also been developed, and excellent surveys are provided in [1, 19].

Although several dynamic load balancers have been proposed for multiprocessor platforms [3, 9, 13, 19, 20], most of them are inadequate for adaptive mesh applications because they lack a global view of system loads across processors. Furthermore, job migration in such approaches does not take into account the structure of the adaptive grid. This motivates our present work. In this paper, we overcome these deficiencies by proposing a novel, dynamic load balancer which makes use of a symmetric broadcast network (SBN) as a robust and topology-independent communication pattern among processors [6]. Section 2 describes this SBN-based load balancing algorithm. Our earlier experiments with synthetic loads [5] have demonstrated that such an SBN strategy achieves superior performance when compared to other popular techniques such as Random, Gradient, Receiver Initiated, Sender Initiated, and Adaptive Contracting.

The SBN-based load balancing algorithm provides an architecture-independent solution in that it generates portable codes which can be run without modification on any parallel/distributed platform. This is because typical communication patterns such as mesh, hypercube, tree, and torus can be embedded efficiently within the SBN topology. It is true that the proposed load balancing scheme in its current form may not be optimal for a given architecture; however, it can be made so by fine tuning the algorithm and properly mapping it on the machine by utilizing its hardware specifications.

Recently, experiments that measure the effectiveness of load balancing adaptive meshes have been presented in [2, 16] using an automatic portable environment, called PLUM [15], developed at NASA Ames Research Center. PLUM uses a novel strategy for load balancing which consists of two separate phases: repartitioning and remapping. A brief overview of PLUM, and a description of its salient differences with the SBN-based load balancer are given in Section 3.

We have conducted several experiments on an IBM SP2 to compare the performance of the SBN-based load balancer to that of PLUM. The results, presented in Section 4, demonstrate that the SBN-based algorithm achieves excellent load balance, and that the redistribution cost is significantly lower than those obtained under PLUM when using two state-of-the-art partitioners, PMeTiS [11] and DMeTiS [17]. However, the edge cut percentages are higher than those for PMeTiS, indicating that the SBN strategy reduces the redistribution cost at the expense of greater communication. In many adaptive mesh applications where the data redistribution cost dominates the processing and communication cost [15, 16, 18], this is an acceptable trade-off.

2 SBN-Based Load Balancer

Our proposed SBN-based load balancer, targeted for adaptive mesh computations, can be classified as: (i) *adaptive*, since processing automatically adjusts to the allocated workload; (ii) *decentralized*, since load balancing can be initiated by any processor in the system and is shared by all; (iii) *stable*, since excessive load balancing traffic does not burden the network; and (iv) *effective*, since system performance does not degrade due to load balancing activities. In this section, we give the definition of an SBN, and present the SBN-based load balancing algorithm. We also describe a pre-partitioner that can optionally be used to assign subdomains to the individual processors before each adaptation step.

2.1 SBN Definition

A *symmetric broadcast network* (SBN), first presented in [6], defines a (logical or physical) communication pattern among the P processors in a multicomputer system. It is defined as follows.

Definition 1 *An SBN(d) of dimension $d \geq 0$, is a $(d+1)$-stage interconnection network with $P = 2^d$ processors in each stage, and can be constructed recursively. A single processor forms the basis network SBN(0). For $d > 0$, SBN(d) is obtained from a pair of SBN($d-1$)s by (i) relabeling the processors in the second SBN($d-1$) from 2^{d-1} to $2^d - 1$; (ii) incrementing the identifiers of the existing stages by one and*

creating a new stage 0 containing processors 0 to $2^d - 1$; (iii) connecting processor i in stage 0 to processor $j = (i + P/2) \bmod P$ of stage 1; and (iv) connecting processor j in stage 1 to the processor in stage 2 (if present) which was the stage 0 successor of processor i in SBN$(d-1)$.

Fig. 1(a) illustrates how an SBN(2) is recursively constructed from two SBN(1)s, while Fig. 1(b) shows the construction of an SBN(3) from two SBN(2)s.

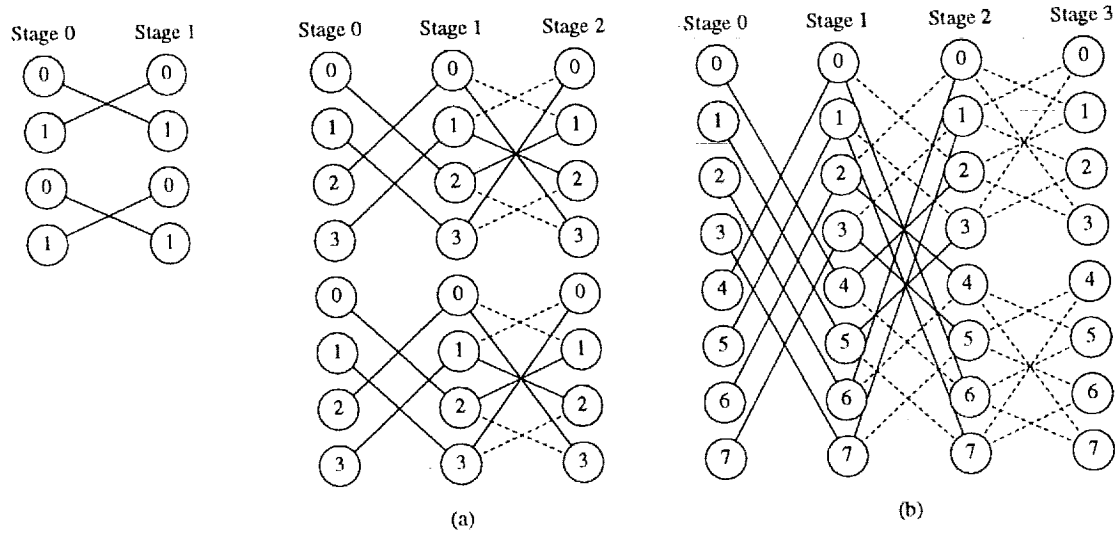

Figure 1: (a) Construction of SBN(2) from a pair of SBN(1)s, and (b) SBN(3) from a pair of SBN(2)s. The new connections are shown by solid lines and the original connections by dashed lines.

Note that an SBN(d) defines unique communication patterns (or broadcast trees) among the processors in the network. In other words, for any root processor x at stage 0, where $0 \leq x < P$, there exists a unique broadcast tree T_x of height $d = \log P$ such that each of the 2^d processors appears exactly once. Furthermore, the SBN communication pattern for x can be derived from the template broadcast tree with processor 0 as the source [5]. The predecessor and successors of each processor are also uniquely defined by specifying the root and the communication stage. Finally, SBN communication patterns can be efficiently embedded into different parallel architectures in a topology-independent manner [4, 7].

2.2 Proposed Load Balancing Algorithm

Our SBN-based load balancer adapts its behavior according to the system load. Under heavy (light) load, the balancing activity is primarily initiated by processors that are lightly (heavily) loaded, and is controlled by two *system load thresholds*, `MinTh` and `MaxTh`. Performance is influenced by the choice of values for `MinTh` and `MaxTh`. If `MinTh` is too small, a processor could become idle before receiving additional jobs for processing. On the other hand, a large value of `MinTh` could trigger unnecessary balancing activity. Similarly, if `MaxTh` is too small, an excessive number of jobs will be migrated; if too large, jobs will not be adequately migrated under light system loads. Moreover, once there is sufficient load in the system, very little load balancing activity should be required.

The load balancer processes two types of messages: (i) load balancing messages and (ii) job distribution messages. A load balancing message is broadcast when a processor p determines that its *weighted queue length* `QWgt`(p) < `MinTh`. Such messages are also broadcast if `QWgt`(p) > `MaxTh`, or if distribution of excess jobs causes other processors to exceed `MaxTh`. As the load balancing message passes from one processor to another, the average *weighted system load*, `WSysLL`, is computed.

Job distribution messages are used to distribute jobs when $\texttt{QWgt}(p) > \texttt{MaxTh}$. They are also used to complete the load balancing process. After the WSysLL value is calculated, a distribution message is broadcast through the SBN so that jobs are routed to lightly-loaded processors and the system control variables (MinTh, MaxTh, and WSysLL) can be globally updated. As a result, all processor workloads are balanced. To reduce message traffic, a processor does not initiate additional load balancing activity until all the previous messages that have passed through it have been completely processed.

Note that it is possible to encounter a situation when there are so many jobs in the system that at least one processor will have its MaxTh value exceeded. This would lead to thrashing, where jobs are unnecessarily routed back and forth among processors. To prevent this situation, if a processor at the last SBN stage determines that its MaxTh has exceeded, it triggers a load balancing message instead of distributing the excess load. As a result, WSysLL and MaxTh are globally recomputed.

Let us now discuss the various parameters and implementation details involved in the SBN-based load balancer. These parameters are necessary to provide a global view of the system and make the SBN approach effective for adaptive mesh applications.

2.2.1 Weighted Queue Length and System Load

The queue length (computation time) of a processor p is not an accurate estimate of the amount of time required to complete its work, particularly in applications where the mesh is adapted. To achieve a better load balance, we define a new metric called weighted queue length, $\texttt{QWgt}(p)$, that also considers the communication and redistribution costs. Let $\texttt{Wgt}^v$ be the computational cost to process a vertex v, $\texttt{Comm}_p^v$ be the communication cost to interact with the vertices adjacent to v but whose data sets are not local to p, and $\texttt{Remap}_p^v$ be the redistribution cost to copy the data set for v to p from another processor. Then

$$\texttt{QWgt}(p) = \sum_{v \text{ assigned to } p} (\texttt{Wgt}^v + \texttt{Comm}_p^v + \texttt{Remap}_p^v).$$

Clearly, if the data set for v is already assigned to p, no redistribution cost is incurred, i.e., $\texttt{Remap}_p^v = 0$. Similarly, if the data sets of all the vertices adjacent to v are already assigned to p, there is no communication cost, i.e., $\texttt{Comm}_p^v = 0$.

The weighted system load, WSysLL, is computed 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.

2.2.2 Prioritized Vertex Selection

When selecting vertices to be processed, the SBN-based load balancer utilizes the underlying structure of the adaptive mesh to defer execution of boundary vertices as long as possible since they could be migrated for more efficient execution. Thus, selection of the queued vertex to be processed next is based on the goal that the overall edge cut of the adapted mesh is minimized. A priority min-queue is maintained for this purpose, where the priority of a vertex v in processor p is given by $(\texttt{Comm}_p^v + \texttt{Remap}_p^v)/\texttt{Wgt}^v$. Therefore, vertices with no communication and redistribution costs are processed first, while those with high communication or redistribution overhead relative to their computational weight are executed last. Conceptually, internal vertices are processed before those on partition boundaries.

2.2.3 Differential Edge Cut

To balance the system load among processors, an optimal policy for vertex migration needs to be established. When vertices are being moved between processors, assume that processor p is about to reassign some of its vertices to another processor q. The SBN-based load balancer running on p randomly picks a subset of vertices from those queued locally. For the experiments reported in this paper, picking a subset of ten vertices worked best. This random procedure reduces the vertex selection overhead since a sorted list of vertices (by migration priority) does not have to be maintained. The motivation was not to find the absolute best vertex to migrate, but rather to identify a vertex that would improve the edge cut as well as the load balance when moved.

For each selected vertex v, the differential edge cut[1], $\Delta\texttt{Cut}(v)$, is calculated as

$$\Delta\texttt{Cut}(v) = \texttt{Remap}_q^v - \texttt{Remap}_p^v + \texttt{Comm}_q^v - \texttt{Comm}_p^v.$$

The parameters $\texttt{Remap}_p^v$ and $\texttt{Remap}_q^v$ will either be zero or equal to the redistribution cost of moving the data for v from p to q. As an example, let $p = 3$ and $q = 6$. Assuming that the data for v reside on $p = 1$ and its redistribution cost is 8, then $\texttt{Remap}_p^v = \texttt{Remap}_q^v = 8$. On the other hand, if the data for v resides on $p = 3$, then $\texttt{Remap}_p^v = 0$ but $\texttt{Remap}_q^v = 8$.

A negative $\Delta\texttt{Cut}(v)$ value indicates a reduction in communication and redistribution costs if v is migrated from p to q; hence, migration of vertices with the largest absolute reduction in these costs is favored. Once the differential edge cut values are calculated for all the randomly chosen vertices, the vertex v' with the smallest value is chosen for migration. Next, following a breadth-first search, the SBN load balancer selects the vertices adjacent to v' that are also queued locally for processing on p. The breadth-first search stops either when no adjacent vertices are queued for local processing at p, or if a sufficient number of vertices have been found for migration. If more work needs to be transferred out of p, another subset of vertices are randomly chosen and the procedure is repeated. This migration policy therefore strives to maintain or improve the cut size during the execution of the load balancing algorithm.

2.2.4 Data Redistribution Policy

The redistribution of data is performed in a lazy manner. In other words, the data set for a vertex v in processor p is not moved to another processor q until the latter is about to execute v (q notifies p when this happens). Furthermore, the data sets of all vertices adjacent to v that are also assigned to q are migrated with the data set of v. This policy greatly reduces the redistribution and communication costs by avoiding multiple data migrations, and having resident on q all adjacent vertices of v while v is being processed by q.

Data migration is implemented by broadcasting a job distribution message when a vertex is about to be processed and its corresponding data set is not resident on the local processor. A locate-message is then broadcast to indicate the new location of the data set, so that all processors can update their records. This policy is expected to maximize the number of adjacent vertices that are local when a given vertex is processed. Hence, by considering the underlying mesh structure, the communication overhead is reduced.

2.3 An Illustrative Example

Fig. 2 illustrates the SBN-based load balancer just described. It shows a mesh of 16 vertices and 20 edges that is partitioned among four processors, $P0$ through $P3$. For each vertex, the processing and redistribution costs are represented as a two-tuple. Adjacent vertices are connected by edges which are labeled with the associated communication cost, provided the data sets for the two vertices reside on different processors when either one is processed.

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

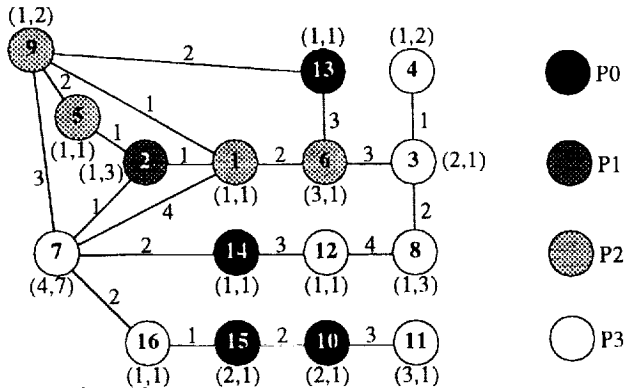

Figure 2: An example to illustrate the SBN-based load balancer.

Table 1 shows the $\texttt{Wgt}^v$, $\texttt{Comm}_p^v$, and $\texttt{Remap}_p^v$ values for each vertex v, under the current vertex-to-processor assignment. We assume that the data for vertex 7 is resident on $P1$, the data for vertex 10 is on $P2$, while the data for vertices 9, 11, and 16, are resident on $P0$. The data sets for the remaining vertices reside on the corresponding processor to which they are assigned. Table 1 also shows the $\texttt{QWgt}(p)$ values for each processor p, as defi ned is Section 2.2.1. The weighted system load, WSysLL, for this example is 24.

Table 1: Various Costs for Each Vertex v, and the Weighted Queue Length for Each Processor p

processor p	$P0$				$P1$	$P2$				$P3$						
vertex v	10	13	14	15	2	1	5	6	9	3	4	7	8	11	12	16
$\texttt{Wgt}^v$	2	1	1	2	1	1	1	3	1	2	1	4	1	3	1	1
$\texttt{Comm}_p^v$	0	3	5	2	2	6	3	6	5	3	0	12	0	3	3	3
$\texttt{Remap}_p^v$	1	0	0	0	0	0	0	0	2	0	0	7	0	1	0	1
$\texttt{QWgt}(p)$	17				3	28				46						

If we assume that $\texttt{MinTh} = 10$, processor $P1$ is clearly underloaded. According to the SBN communication pattern shown in Fig. 1(a), $P1$ sends a load balancing request to $P3$. Upon receiving it, $P3$ determines which of its vertices to transfer to $P1$ so that their loads will be equidistributed. Let us step through the process of selecting the fi rst vertex to migrate, using the differential edge cut described in Section 2.2.3. The $\Delta\texttt{Cut}(v)$ values of the vertices v currently assigned to $P3$ are shown in Table 2. Vertex 7 is found to be optimal for migration to $P1$, yielding $\texttt{QWgt}(P3) = 23$ and $\texttt{QWgt}(P1) = 18$. The new value of WSysLL is

Table 2: Differential Edge Cut for Each Vertex v in Processor $P3$ if Migrated to $P1$

vertex v	3	4	7	8	11	12	16
$\texttt{Remap}_{P1}^v$	1	2	0	3	1	1	1
$\texttt{Remap}_{P3}^v$	0	0	7	0	1	0	1
$\texttt{Comm}_{P1}^v$	6	1	11	6	3	7	1
$\texttt{Comm}_{P3}^v$	3	0	12	0	3	3	3
$\Delta\texttt{Cut}(v)$	4	3	−8	9	0	5	−2

22, which reflects a reduction in the total system load. For this example, additional vertex migration is not required.

2.4 SBN Pre-Partitioner

The SBN-based load balancing algorithm is designed to run dynamically without the need for a separate partitioning process. This is a significant advantage over existing approaches where processing is temporarily suspended when processor loads become unbalanced. During the suspension, vertices are reassigned and the corresponding data sets are remapped. The asynchronous nature of the SBN strategy also allows the computational, communication, and redistribution phases to be overlapped, leading to further reductions in the overall execution time. Traditional methods [15, 19, 20] cannot achieve this overlap easily because these phases are processed sequentially.

To test the behavior of the SBN technique, we implemented a pre-partitioner which can optionally run prior to each mesh adaptation phase. We wanted to determine whether running a front-end partitioner has any significant benefit on the resulting communication and/or redistribution overhead. This pre-partitioner is unique in that it partitions based on $\texttt{QWgt}(p)$ values, which take into consideration all three factors of computation, communication, and redistribution. This is a stronger requirement than that considered in almost all other approaches [8, 11, 17, 21], where the mesh is partitioned to equalize the total computational cost while minimizing the total number of cut edges. Such methods could result in significant idle time during processing if only a few processors incurred most of the communication overhead.

The pre-partitioner differs from the partitioning capabilities inherent in the SBN-based load balancer in that multiple iterations are performed to find an optimal P-way partition. Here, an *iteration* is defined as a sequence of vertex reassignments from one processor to another. During an iteration, each vertex can be reassigned at most once. Reassignments are made so that vertices in processor p with $\texttt{QWgt}(p) > \texttt{WSysLL}$ are assigned to the processor q with the minimum $\texttt{QWgt}(q)$ value. Each vertex to be reassigned is adjacent to a random subset of vertices chosen from and belonging to q. First, the $\Delta\texttt{Cut}(v)$ values are computed for all adjacent vertices v assigned to processors other than q. As described in Section 2.2.3, the non-local adjacent vertex v' with the smallest $\Delta\texttt{Cut}(v')$ is added to the set of vertices assigned to q. In addition, a breadth-first search is performed on the vertices adjacent to v' that are not assigned to q but to p such that $\texttt{QWgt}(p) > \texttt{WSysLL}$. These vertices are also assigned to q.

The pre-partitioner is initially set to execute a fixed number of iterations (four for the experiments in this paper). However, additional iterations are performed if a new minimum WSysLL is achieved. At the end of each iteration, the load imbalance factor $\texttt{QWgt}(r)/\texttt{WSysLL}$ for the processor r with the largest value of $\texttt{QWgt}(r)$ is computed. If this factor is greater than a specified threshold (1.75 in our experiments), the Kernighan-Lin refinement procedure [12] is invoked to further reduce WSysLL. Note that the data associated with each vertex is not migrated after the pre-partitioning process is completed. Instead, actual data movement takes place during mesh adaptation as vertices are processed with SBN load balancing in effect.

3 PLUM Framework

We experimentally compare the performance of our SBN-based load balancer with PLUM [15], a portable and parallel load balancing framework for adaptive unstructured grids. In PLUM, when processor workloads become unbalanced due to adaptation, the mesh is repartitioned and the subgrids reassigned to the processors. If the estimated remapping cost exceeds the expected computational gain, execution continues without remapping. Otherwise, the grid is remapped among the processors before the computation is resumed. For the sake of completeness, a brief description of the important features of PLUM is given below.

3.1 Reusing the Initial Graph

PLUM repeatedly utilizes the initial mesh for the purpose of load balancing. The computational weight, $\texttt{Wgt}^v$, of a vertex v in the corresponding dual graph, is the number of leaf elements in the refinement tree because only those elements with no children participate in the numerical computation. The redistribution cost, $\texttt{Remap}^v$, is the total number of elements in the refinement tree because all descendents of the root element must be moved from one partition to another when the load is to be rebalanced. Lastly, the communication cost, $\texttt{Comm}^e$, of a dual graph edge e, is set to the number of corresponding faces in the computational mesh. These weights are used to determine an optimal partitioning that achieves balanced workloads among processors, to minimize the resulting communication, and to optimize the data movement cost.

3.2 Parallel Mesh Repartitioning

PLUM can use any general-purpose partitioner to rebalance processor workloads after a mesh adaptation. In [2], PMeTiS [11] and DMeTiS [17] were used. Both partitioners are parallelized and highly optimized for maximum efficiency, and have proven effective for adaptive grids. DMeTiS is a diffusive scheme designed to modify existing partitions, while PMeTiS is a global from-scratch partitioner that makes no assumptions on how the mesh is initially distributed. Both are multilevel algorithms that operate in three phases: (i) a coarsening phase, where the original mesh is reduced by collapsing adjacent vertices to a sufficiently small mesh; (ii) a partitioning phase, where the coarsened mesh workload is balanced among the processors and the edge cut size is minimized; and (iii) a projection phase, where the partitioned mesh is gradually restored to its original size.

DMeTiS and PMeTiS differ mainly in how they perform the partitioning phase. DMeTiS uses a directed 2-norm minimization algorithm [10] which provides a global picture of the existing mesh. Vertices in heavily-loaded partitions that are adjacent to neighbors in more lightly-loaded partitions are randomly visited. The diffusion process computes a flow value for possible reassignment to neighboring partitions. If the flow value relative to the vertex weight is high, the vertex is reassigned. This process continues until the partition is balanced or no further progress can be made. If a balanced partitioning cannot be achieved at the current level of the mesh, it is projected to the next finer level and the partitioning process is repeated. PMeTiS, on the other hand, utilizes a greedy recursive bisection algorithm to create a partition of the graph from scratch. The time complexity for both algorithms is minimal since the partitioning is performed on a coarse graph containing a small number of vertices and edges.

3.3 Processor Remapping

The goal of processor reassignment is to find a mapping between partitions and processors that minimizes the cost of data redistribution. To achieve this, PLUM computes a similarity matrix S, where entry S_{ij} is the sum of the $\texttt{Remap}^v$ values of all vertices in the new partition j that already reside on processor i. Various cost functions [16] are usually needed to solve the reassignment problem using S for different machine architectures. In [2], an efficient heuristic algorithm was developed to minimize the volume of data that is moved among the processors. This algorithm has been shown to be no worse than twice the optimal performance.

3.4 Cost Model

Predicting the expected redistribution overhead is difficult because of the large number and complexity of the costs involved. For example, it includes the cost for rebuilding internal data structures and updating shared boundary information. Furthermore, the total redistribution cost depends on the architecture and on the many-to-many communication patterns used by the remapper. In PLUM, the equation $\gamma \times \texttt{MaxSR} + O$ is used

to model the total cost [2, 16]. Here, γ represents the computation and communication overhead to process each redistributed element, `MaxSR` is the maximum number of elements sent and received by any processor, and O is the predicted sum of all other constant overheads such as data compaction, communication latency, and barrier synchronization. A least squares fi t can be used to approximate γ and O for various architectures, while `MaxSR` is computed from the similarity matrix S.

Once the redistribution cost is computed, it can be compared with the expected computational gain achieved by reducing the load imbalance among the processors. If the computational gain is larger than the redistribution cost, the new partitioning and mapping are accepted. Otherwise, the computation is resumed on the unbalanced mesh.

3.5 Differences with SBN-Based Load Balancer

The SBN load balancing algorithm differs from PLUM in several ways. Here we itemize the salient differences:

- Processing is temporarily halted under PLUM while the load is balanced. During the suspension, a new partitioning is generated and data is redistributed among the processors. The SBN approach, on the other hand, allows processing to continue asynchronously with load balancing. This feature allows the possibility of utilizing latency-tolerant techniques to hide communication and redistribution costs during processing.
- With PLUM, the suspension of processing and subsequent repartitioning does not guarantee an improvement in the quality of load balance. If it is determined that the estimated remapping cost exceeds the expected computational gain, processing continues using the original mesh assignment. This could result in unnecessary idle time. In contrast, the SBN approach, when active, always reduces the execution time for the application.
- PLUM redistributes all necessary data to the appropriate processors before processing is restarted. SBN, however, distributes work in a lazy manner, i.e., data is migrated to a processor only when it is ready to process the data. In this way, some of the redistribution and communication overhead can be avoided.

4 Experimental Study

The SBN-based load balancing algorithm has been implemented using MPI on the wide-node IBM SP2 located at NASA Ames Research Center, and tested with actual workloads obtained from an adaptive unstructured-grid calculation.

4.1 Test Case

The computational mesh used for the experiments reported in this paper simulates an unsteady environment where the adapted region is strongly time-dependent. This goal is achieved by propagating a simulated shock wave through the initial mesh as shown in Fig. 3. The test case is generated by refi ning all elements within a cylindrical volume moving left to right across the domain, while coarsening previously-refi ned elements in its wake. Performance is measured at nine successive adaptation levels, during which the weighted sum of the vertices increased from 50,000 to 1,833,730. The levels shown in Tables 3 and 4 indicate successive positions of the shock wave as it progresses through the cylindrical volume. This test case was chosen so that results could be compared with those compiled in [2] under the PLUM environment.

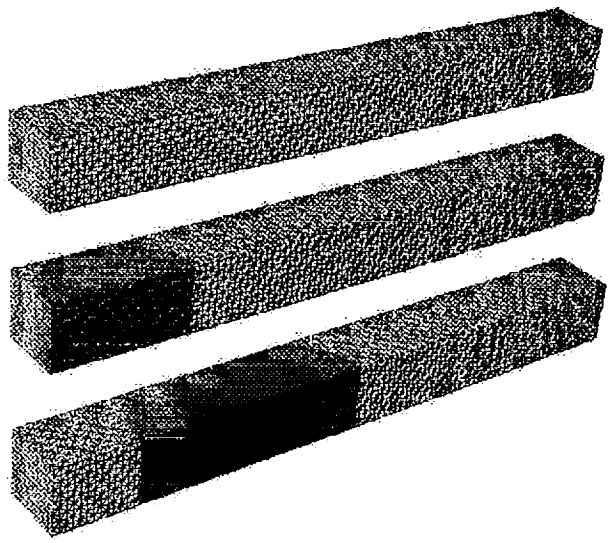

Figure 3: Initial and adapted meshes (after levels 1 and 5) for the simulated unsteady experiment.

4.2 Performance Metrics

The following metrics were chosen to evaluate the effectiveness of the SBN-based load balancer when processing an unsteady adaptive mesh. Recall that v denotes a vertex to be processed and P is the total number of processors.

- **Cut percentage:** The runtime interaction between adjacent vertices residing on different processors is represented by this metric as:

$$\texttt{Cut\%} = 100 \times \sum_{p \in P} \sum_{v \text{ assigned to } p} \texttt{Comm}_p^v \Bigg/ \sum_{e \text{ in mesh}} \texttt{Comm}^e ,$$

  where $\texttt{Comm}^e$ is the weight of edge e in the adaptive mesh. The `Cut%` value should be as small as possible. The `PrePartCut%` (see Table 3) is the projection of the mesh edge cut before running the SBN pre-partitioner. On the other hand, `PreExecCut%` computes the mesh edge cut immediately before processing a mesh adaptation level, while `PostExecCut%` is the actual cut realized after processing the given adaptation level.

- **Maximum redistribution cost:** The goal of this metric is to capture the total cost of packing and unpacking data, separated by a barrier synchronization. Since a processor can either be sending or receiving data, the overhead of these two phases is modeled as a sum of two costs as:

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

  Since `MaxSR` pertains to the processor that incurs the maximum remapping cost, a reduction in the total data redistribution overhead can be guaranteed by minimizing `MaxSR`.

- **Load imbalance factor:** This metric is the ratio of the work on the most heavily-loaded processor to the average load across all processors, and is formulated as:

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

The LoadImb factor should be as close to unity as possible.

4.3 Summary of Results

Table 3 presents performance results of processing the adaptive mesh using the SBN-based load balancer, with and without the SBN pre-partitioner running between adaptations. Table 4 charts the results achieved using the PMeTiS and DMeTiS partitioners within the PLUM environment. Note that Table 4 does not contain results corresponding to all the processor sets shown in Table 3. We have included only those values that were available to us.

The LoadImb factors are not shown in Table 3 since they were consistently between 1.00 and 1.02, indicating that the quality of load balance with the SBN-based approach was extremely high. In contrast, this factor was respectively 1.04 and 1.59 for $P = 32$ using PMeTiS and DMeTiS under the PLUM environment (see Table 4). Obviously, LoadImb is poorer for DMeTiS because of its diffusive nature.

Results show that the SBN PostExecCut%, when using the pre-partitioner, is more than double compared to those reported by PMeTiS (21.29 in Table 3 vs. 10.94 in Table 4, for $P = 32$). The difference is almost negligible when compared to the results obtained with DMeTiS (20.22 in Table 4). This could reflect the effectiveness of the partitioners being used rather than whether the SBN-based load balancer would always produce higher communication costs. Note that PostExecCut% is about 1.5 times higher when the SBN pre-partitioner is not active (see Table 3). This implies that it may be useful to initially partition the mesh to compute a starting point for subsequent SBN load balancing when high communication cost is a critical factor.

The MaxSR metric is proportional to the redistribution cost incurred while processing the adaptive mesh. The SBN lazy approach to migration of vertex data sets produces signifi cantly lower values than those achieved by PMeTiS or DMeTiS under PLUM. For example, when $P = 32$, Table 3 shows MaxSR $= 28,031$ without the SBN pre-partitioner, which is signifi cantly less than the corresponding values in Table 4 (63,270 for PMeTiS and 62,542 for DMeTiS). However, when the SBN pre-partitioner is used with the load balancer, the MaxSR value increases (see Table 3). Thus, there is a trade-off here: the pre-partitioner reduces runtime interprocessor communication at the expense of a higher data redistribution cost. Finally, by comparing PrePartCut% and PreExecCut% in Table 3, observe that Cut% degrades as the pre-partitioner executes. This result is consistent with the observations drawn from the PLUM experiments [2].

In conclusion, our experiments demonstrate that using the SBN pre-partitioner produces lower communication costs but higher data remapping costs. Although the pre-partitioner may be of limited value for those adaptive mesh applications where remapping costs dominate communication costs, it could be useful in scenarios where reducing the communication cost is more important. Overall, these performance results demonstrate that the proposed SBN-based dynamic load balancer is effective for processing adaptive mesh problems by providing a global workload view across processors. In many mesh applications where the cost of data redistribution dominates the cost of communication and processing, the SBN-based algorithm would be preferred.

4.4 Complexity Analysis

In this section, we analyze the overhead associated with the execution of the SBN-based load balancer while processing the adaptive computational mesh. The overhead has four components: (i) selecting the next vertex to be processed; (ii) selecting the set of vertices to be migrated; (iii) processing to determine if

Table 3: Performance Results using the SBN-Based Load Balancer with (without) Pre-Partitioning

P	Level	PrePartCut%		PreExecCut%		PostExecCut%		MaxSR	
2	1	0.09	—	1.76	(0.09)	0.95	(4.64)	9,606	(6,974)
	2	1.21	—	2.82	(3.14)	1.60	(6.18)	41,926	(30,538)
	3	0.59	—	3.36	(5.36)	2.60	(6.08)	178,631	(57,724)
	4	3.28	—	4.00	(3.93)	2.31	(3.86)	118,679	(20,646)
	5	2.94	—	3.02	(2.91)	2.39	(5.32)	112,437	(76,893)
	6	4.36	—	3.88	(2.33)	2.93	(4.62)	87,517	(103,544)
	7	2.76	—	2.53	(2.23)	1.78	(5.86)	75,925	(140,904)
	8	0.51	—	3.14	(2.83)	2.08	(6.14)	223,160	(153,735)
	9	2.55	—	2.87	(3.10)	2.18	(6.89)	103,772	(129,374)
	Average	2.03	—	3.04	(2.88)	2.09	(5.51)	105,739	(80,037)
4	1	2.26	—	3.67	(2.26)	2.58	(8.15)	6,937	(4,078)
	2	3.37	—	4.11	(7.22)	3.13	(10.01)	24,382	(26,187)
	3	3.60	—	6.03	(9.44)	4.96	(11.69)	81,348	(64,110)
	4	5.73	—	5.51	(9.16)	4.56	(9.48)	85,345	(46,406)
	5	6.23	—	6.58	(6.60)	4.89	(11.86)	101,070	(149,042)
	6	6.25	—	6.29	(9.83)	5.12	(10.89)	51,018	(94,269)
	7	5.95	—	8.22	(6.58)	6.72	(8.00)	145,850	(50,337)
	8	7.45	—	8.36	(2.79)	6.86	(15.31)	92,430	(170,408)
	9	6.05	—	9.63	(11.53)	4.99	(11.48)	69,413	(85,152)
	Average	5.21	—	6.49	(7.27)	4.87	(10.76)	73,088	(76,665)
8	1	6.66	—	7.16	(6.66)	6.05	(10.77)	6,939	(2,518)
	2	7.60	—	7.56	(13.93)	6.17	(14.98)	22,833	(11,109)
	3	7.85	—	8.48	(15.11)	7.33	(18.16)	90,132	(46,088)
	4	7.78	—	17.35	(14.65)	14.67	(15.83)	139,439	(53,032)
	5	12.64	—	12.19	(11.09)	11.58	(16.48)	138,671	(69,583)
	6	7.97	—	11.19	(11.02)	9.88	(15.91)	123,433	(85,982)
	7	12.09	—	11.81	(13.75)	10.74	(18.13)	130,199	(105,946)
	8	12.39	—	10.99	(12.84)	9.93	(19.51)	123,223	(28,974)
	9	7.93	—	10.06	(15.34)	8.90	(17.35)	158,867	(80,477)
	Average	9.21	—	10.75	(12.71)	9.47	(16.35)	103,748	(53,745)
16	1	15.36	—	11.48	(15.36)	11.01	(20.61)	5,647	(1,767)
	2	13.15	—	11.71	(24.82)	11.37	(25.56)	26,263	(7,259)
	3	12.89	—	13.02	(24.40)	12.59	(27.45)	107,173	(36,031)
	4	8.38	—	22.60	(20.60)	21.39	(22.77)	209,028	(43,943)
	5	17.08	—	14.05	(16.11)	14.28	(24.27)	122,902	(71,736)
	6	14.66	—	13.14	(17.83)	12.55	(22.28)	100,962	(66,211)
	7	13.08	—	21.37	(19.75)	19.12	(25.00)	135,900	(55,361)
	8	16.63	—	13.57	(17.83)	14.76	(25.30)	126,161	(64,796)
	9	12.13	—	23.36	(17.87)	21.57	(21.59)	102,203	(74,316)
	Average	13.71	—	16.03	(19.40)	15.41	(23.87)	104,027	(46,824)
32	1	21.59	—	13.57	(21.59)	15.50	(26.74)	3,764	(1,184)
	2	18.20	—	14.49	(30.35)	14.28	(32.32)	10,784	(4,387)
	3	14.59	—	20.25	(30.06)	19.78	(34.04)	53,423	(8,445)
	4	13.43	—	21.14	(27.28)	23.35	(31.43)	154,009	(41,783)
	5	15.95	—	28.07	(21.35)	29.09	(29.40)	196,821	(42,843)
	6	19.94	—	21.65	(24.04)	22.19	(29.42)	117,254	(42,688)
	7	17.07	—	23.23	(22.35)	23.82	(30.45)	90,404	(41,347)
	8	18.44	—	17.62	(20.59)	20.30	(30.48)	90,322	(37,006)
	9	14.58	—	23.74	(22.19)	23.31	(29.43)	116,354	(32,594)
	Average	17.09	—	20.42	(24.42)	21.29	(30.41)	92,571	(28,031)

Table 4: Performance Results using PMeTiS (DMeTiS) under the PLUM Environment

P	Level	PreExecCut%		PostExecCut%		MaxSR		LoadImb	
16	1	3.16		4.38		10,088		1.02	
	2	5.34		7.20		25,875		1.02	
	3	7.27		9.71		58,887		1.03	
	4	5.24		8.62		134,808		1.03	
	5	5.77		8.17		153,154		1.04	
	6	4.70		8.06		122,151		1.02	
	7	4.47		8.45		159,037		1.02	
	8	5.31		7.97		132,987		1.01	
	9	4.18		7.75		130,824		1.01	
	Average	5.05		7.81		103,090		1.02	
32	1	4.78	(4.65)	6.45	(15.70)	5,097	(5,047)	1.01	(1.88)
	2	7.56	(19.26)	10.05	(20.50)	16,758	(17,393)	1.02	(2.12)
	3	10.28	(21.14)	13.13	(25.26)	39,565	(44,413)	1.05	(2.12)
	4	8.14	(17.13)	11.60	(28.21)	73,074	(99,232)	1.06	(1.87)
	5	7.59	(29.08)	11.13	(26.46)	92,581	(97,280)	1.05	(1.68)
	6	6.51	(25.31)	11.60	(24.38)	82,751	(86,204)	1.06	(1.41)
	7	6.66	(20.55)	11.43	(14.17)	88,642	(78,312)	1.03	(1.11)
	8	6.88	(10.04)	11.39	(13.08)	91,301	(72,474)	1.05	(1.05)
	9	6.19	(9.41)	11.66	(14.18)	79,662	(62,522)	1.04	(1.05)
	Average	7.18	(17.40)	10.94	(20.22)	63,270	(62,542)	1.04	(1.59)

load balancing is necessary; and (iv) load balancing and job distribution messages (communication) among processors. Where possible, both analytical formulas and experimental data are presented.

The vertex v to be processed next is selected using a priority min-queue. Let V_p be the set of vertices to be processed at a given processor p, and E_p be the set of all internal and border edges that are adjacent to the vertices in V_p. Heap operations like create and insert/delete-min require $O(V_p)$ and $O(\log V_p)$ time, respectively. The (non-standard) removal operation can be implemented in $O(\log V_p)$, provided a direct pointer to the entry to be removed is maintained. However, for SBN processing, the sum ($\texttt{Wgt}^v + \texttt{Comm}_p^v + \texttt{Remap}_p^v$) must be computed so that the value of $\texttt{QWgt}(p)$ can be obtained (see Section 2.2.1). Also, ($\texttt{Comm}_p^v + \texttt{Remap}_p^v$)/$\texttt{Wgt}^v$ is needed to correctly control the ordering of the priority min-queue (see Section 2.2.2). Each of these calculations requires $O(\delta_v)$ time, where δ_v is the degree of v. Therefore, the SBN priority min-queue (heap) creation requires $O(V_p + \sum_{v \in V_p} \delta_v) = O(V_p + E_p)$ time. Similarly, each heap insertion, delete-min, and removal operation completes in $O(\log V_p + \delta_v)$ time.

SBN vertex migration involves first selecting a random set, R, of vertices from those queued locally. The vertex, $v' \in R$, with the smallest $\Delta\texttt{Cut}(v')$ value is chosen for migration (see Section 2.2.3). Each $\Delta\texttt{Cut}(r)$ calculation completes in $O(\delta_r)$ time, where $r \in R$. Therefore, the total time required to select the initial vertex for migration is $O(\sum_{r \in R} \delta_r) \approx O(|R| \times \delta_{avg})$, where δ_{avg} is the average degree of a vertex in the mesh. Next, the local queue is searched in a breadth-first manner to choose an additional set, V_{mig}, of vertices for migration with v'. In our experiments, $|V_{mig}|$ averaged less than ten to satisfy the requirements of a load balancing operation. Furthermore, a single search almost always found enough vertices to migrate. Thus, the time required to complete the breadth-first search is $O(|V_{mig}| + \sum_{v \in V_{mig}} \delta_v) \approx O(|V_{mig}| \times (1 + \delta_{avg}))$. Finally, each vertex to be migrated must be removed from the priority min-queue so that they will no longer be considered for local processing. Since $|V_{mig}| + 1$ removal operations are required, the time complexity for this step is $O((|V_{mig}|+1) \times \log V_p + \sum_{v \in V_{mig}} \delta_v) \approx O(|V_{mig}| \times (\log V_p + \delta_{avg}))$. Combining

the above three terms and considering that $|R|$ is a constant, the overall asymptotic time complexity for job migration is $O(|V_{mig}| \times (\log V_p + \delta_{avg}))$.

Each processor must periodically check whether a load balancing operation should be initiated or if messages from other processors need to be processed. If processors check too frequently, the associated overhead could be too high. On the other hand, infrequent checks for load balancing activity could lead to excessive idle time. The following analysis can be used to minimize this overhead without signifi cantly increasing processor idle time. If f is the frequency of a processor checking for load balancing activity, the average response time to process a message is $2/f$. Each time the SBN-based load balancer is invoked, balancing and distribution messages pass through $3 \log P$ communication stages. Therefore, the total response time to balance the system load is $(6 \log P)/f$. If J_{avg} is the average number of jobs processed per unit time, the `MinTh` threshold should be set such that load balancing will be triggered when $\texttt{QWgt}(p) < \texttt{MinTh}$, to avoid excessive idle time (see Section 2.2). In other words, $\texttt{MinTh} \geq \lceil 6 \log P \times J_{avg}/f \rceil$.

The communication overhead due to message passing is measured experimentally. Table 5 shows the number of Mbytes that were transferred between processors during the load balancing and job distribution phases. The data volumes are also expressed as percentages of the available bandwidth. A wide-node SP2 has a bandwidth of 36 Mbytes/sec and a latency of 40 μsecs. As expected, the cost of workload migration is signifi cantly larger than the cost of actually balancing the system load. An extrapolation of the results using an exponential curve-fi tting program indicates that parallel speedup will not scale past 128 processors. Most of the overhead is due to the latency associated with transmitting many small messages; however, it is asymptotically sublinear in the total number of processors used. Future research will investigate utilizing latency-tolerant techniques to allow for bulk transfers.

Table 5: Communication Overhead of the SBN-Based Load Balancer

	Load Balancing Phase		Job Distribution Phase	
P	Volume (MBytes)	Bandwidth (%)	Volume (MBytes)	Bandwidth (%)
2	0.342	0.00	3.919	3.67
4	0.150	0.00	7.939	7.44
8	0.463	0.01	25.397	23.79
16	0.581	0.02	30.454	28.53
32	1.550	0.12	38.244	35.83

Table 6 shows the fraction of time spent in the SBN-based load balancer compared to the total execution time of the mesh adaptation application. The three columns in the table correspond to three categories of load balancing activity: (i) time needed to handle load balancing messages, (ii) time needed to migrate vertices from one processor to another, and (iii) time needed to select the next vertex to be processed. Results

Table 6: Percentage Overhead of the SBN-Based Load Balancer

	Balancing	Migration	Vertex
P	Activity	Activity	Selection
2	0.0153	0.0414	0.8490
4	0.0187	0.1069	1.1361
8	0.1245	0.1969	1.9886
16	0.6369	0.2829	2.1145
32	0.1554	0.3774	2.8543

show that processing related to the selection of vertices is the most expensive phase. The SBN algorithm dynamically chooses the next vertex to be processed, depending on specific runtime criteria. Thus, vertex selection is not as efficient as parallel multilevel partitioning. However, the data movement cost in the SBN approach is substantially smaller than that of traditional remapping schemes since it allows processing to continue while the load is dynamically balanced, thereby overlapping processing and migration. Overall, the total overhead of our load balancer is relatively small compared to the time spent processing the mesh.

5 Summary

In this paper, we have described a novel topology-independent approach to solving the dynamic load balancing problem for adaptive meshes. Our thorough experimental investigation with an unstructured adaptive mesh application showed that the proposed SBN-based load balancer achieves a lower redistribution cost than that under the PLUM environment. This was possible by overlapping processing and data migration. However, the communication costs using SBN were significantly higher than those reported under PLUM. Overall, the SBN approach was demonstrated to be a viable option in load balancing dynamic irregular applications.

The SBN-based load balancer is not purely diffusive, in that work is not necessarily migrated to neighboring processors. In fact, a vertex is usually redistributed to a processor that owns an adjacent vertex. While diffusive strategies are fairly common, scratch-remap techniques (similar to that used in PLUM) have also been used successfully to load balance adaptive mesh applications. Our more recent work on the SGI Origin2000 system is consistent with the performance results presented here, showing the portability of the SBN-based load balancing algorithm.

Because of its latency-tolerance feature, it seems natural to evaluate the performance of the SBN approach on a heterogeneous cluster of computers. Another research arena includes strategies to adapt the processing to situations where some of the processors in the network become unavailable during a computation. Such fault tolerance would allow applications to make use of resources that are constantly changing during execution. Finally, the techniques presented here could be applied to other practical applications, such as multimedia image processing and data mining, where load balancing is an important issue. These will be the focus of future research.

Acknowledgements

The work of the first two authors was partially supported by Texas Advanced Research Program Grant Number TARP-97-003594-013 and by NASA Ames Research Center under Cooperative Agreement Number NCC 2-5395.

References

[1] C. Alpert and A. Kahng, "Recent Directions in Netlist Partitioning," *Integration, the VLSI J.*, vol. 19, pp. 1–81, 1995.

[2] R. Biswas and L. Oliker, "Experiments with Repartitioning and Load Balancing Adaptive Meshes," *Grid Generation and Adaptive Algorithms*, IMA vol. 113, pp. 89–111, 1999.

[3] N. Chrisochoides, "Multithreaded Model for the Dynamic Load Balancing of Parallel Adaptive PDE Computations," *Applied Numerical Math.*, vol. 20, pp. 349–365, 1996.

[4] S.K. Das and D.J. Harvey, "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, 1995.

[5] S.K. Das, D.J. Harvey, and R. Biswas, "Adaptive Load Balancing Algorithms using Symmetric Broadcast Networks: Performance Study on an IBM SP2," *Proc. 26th Int'l. Conf. on Parallel Processing*, pp. 360–367, 1997.

[6] S.K. Das and S.K. Prasad, "Implementing Task Ready Queues in a Multiprocessing Environment," *Proc. Int'l. Conf. Parallel Computing*, pp. 132–140, 1990.

[7] S.K. Das, S.K. Prasad, C-Q. Yang, and N.M. Leung, "Symmetric Broadcast Networks for Implementing Global Task Queues and Load Balancing in a Multiprocessor Environment," Technical Report CRPDC-92-1, Dept. of Computer Science, Univ. of North Texas, 1992.

[8] B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs," *Proc. Supercomputing '95*, 1995.

[9] G. Horton, "A Multi-Level Diffusion Method for Dynamic Load Balancing," *Parallel Computing*, vol. 19, pp. 209–229, 1993.

[10] Y.F. Hu and R.J. Blake, "An Optimal Dynamic Load Balancing Algorithm," Technical Report DL-P-95-0-11, Daresbury Laboratory, 1995.

[11] G. Karypis and V. Kumar, "Parallel Multilevel k-Way Partitioning Scheme for Irregular Graphs," Technical Report 96-036, Dept. of Computer Science, Univ. of Minn., 1996.

[12] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Systems Tech. J.*, vol. 49, pp. 291–307, 1970.

[13] G.A. Kohring, "Dynamic Load Balancing for Parallelized Particle Simulations on MIMD Computers," *Parallel Computing*, vol. 21, pp. 683–693, 1995.

[14] N. Mansour, R. Ponnusamy, A.N. Choudhary, and G.C. Fox, "Graph Contraction and Physical Optimization Methods: A Quality-Cost Tradeoff for Mapping Data on Parallel Computers," *Proc. 7th Int'l. Conf. Supercomputing*, pp. 1–10, 1993.

[15] L. Oliker and R. Biswas, "PLUM: Parallel Load Balancing for Adaptive Unstructured Meshes," *J. Parallel and Distributed Computing*, vol. 52, pp. 150–177, 1998.

[16] L. Oliker, R. Biswas, and H.N. Gabow, "Parallel Tetrahedral Mesh Adaptation with Dynamic Load Balancing," *Parallel Computing*, vol. 26, pp. 1583–1608, 2000.

[17] K. Schloegel, G. Karypis, and V. Kumar, "Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes," *J. Parallel and Distributed Computing*, vol. 47, pp. 109–124, 1997.

[18] H. Shan, J.P. Singh, L. Oliker, and R. Biswas, "A Comparison of Three Programming Models for Adaptive Applications on the Origin2000," *Proc. Supercomputing '00*, 2000.

[19] R. Van Driessche and D. Roose, "Load Balancing Computational Fluid Dynamics Calculations on Unstructured Grids," *Parallel Computing in CFD*, AGARD-R-807, pp. 2.1–2.26, 1995.

[20] A. Vidwans, Y. Kallinderis, and V. Venkatakrishnan, "Parallel Dynamic Load Balancing Algorithm for Three-Dimensional Adaptive Unstructured Grids," *AIAA Journal*, vol. 32, pp. 495–505, 1994.

[21] C. Walshaw, M. Cross, and M.G. Everett, "Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes," *J. Parallel and Distributed Computing*, vol. 47, pp. 102–108, 1997.

Author Biographies

SAJAL K. DAS is currently a Professor of Computer Science and Engineering, and since 1999, also the Founding Director of the Center for Research in Wireless Mobility and Networking (CReWMaN) at the University of Texas at Arlington (UTA). Prior to that, he was a Professor of Computer Science at the University of North Texas (UNT), Denton, where he founded the Center for Research in Wireless Computing (CReW) in 1997, and served as the Director of the Center for Research in Parallel and Distributed Computing (CRPDC) during 1995–97. Dr. Das is a recipient of the UNT Student Association's Honor Professor Award in 1991 and 1997 for best teaching and scholarly research; UNT's Developing Scholars Award in 1996 for outstanding research; and UTA's Outstanding Senior Faculty Research Award in Computer Science in 2001. An internationally reputed professor, he has visited numerous universities and research organizations worldwide for collaborative research and seminar talks. He held Visiting Scientist positions at the Council of National Research in Pisa and at the Slovak Academy of Sciences in Bratislava, and was a Visiting Professor at the Indian Statistical Institute in Calcutta. He is frequently invited as a keynote speaker at international conferences and symposia.

His current research interests include mobile computing, wireless networks, QoS in wireless multimedia and mobile Internet, parallel and distributed computing, parallel data structures, and performance modeling and simulation. He has published over 175 research papers in these areas, directed several funded projects, and holds four US patents in wireless moble networks. He received the Best Paper Awards for signifi cant research contributions at MSWIM-2000, MobiCom'99, and PADS'97. Dr. Das serves on the Editorial Boards of four international journals including Subject Area Editor of Mobile Computing for JPDC. He has guest-edited special issues of ACM Wireless Networks, JPDC, and IEEE Transactions on Computers. He served as General Chair of WoWMoM-2000 and PDC-WNMC-2001, General Vice-Chair of MobiCom-2000, HiPC-2000, and HiPC-2001, General Co-Chair of MASCOTS'98, Founding TPC Chair of WoWMoM'98 and WoWMoM'99, Program Vice-Chair of HiPC'99, and TPC member of numerous IEEE and ACM conferences including INFOCOM, MobiCom, and IPDPS. He is also a member of the ACM SIGMOBILE and IEEE TCPP Executive Committees.

DANIEL J. HARVEY is assuming a faculty position in the Department of Computer Science at Southern Oregon University, Ashland, starting in September 2001. Previously, from 1992 to 2001, he was an Assistant Professor at Dallas Baptist University. In August 2001, Mr. Harvey will graduate from the University of Texas at Arlington with a PhD in computer science, where he was nominated for the 2001 UTA Outstanding Doctoral Research Student Award. One of his paper was a fi nalist for the Best Paper Award at the CCGrid2001 conference. His research interests include parallel programming, load balancing, and cluster computing.

Mr. Harvey has extensive experience in industry and was President of an automotive aftermarket value-added reseller for 13 years. He has considerable consulting experience working with fi rms such as Dunn & Bradstreet and Honeywell. In 1999, under a NASA summer grant, he conducted research at NASA Ames Research Center, Moffett Field, California. Mr. Harvey is also active in athletics, completing his 5th marathon at Boston in 1997. He established the Dallas Baptist University cross-country and track programs in 1996, and served as their Head Coach through 2001.

RUPAK BISWAS received the BSc (Honors) in physics (1982) and the BTech in computer science (1985), both from the University of Calcutta, India, and the MS (1988) and PhD (1991) degrees in computer science from Rensselaer Polytechnic Institute, Troy, New York. He is currently a Senior Computer Scientist with NASA Ames Research Center, Moffett Field, California. In the past, he was a Senior Research Scientist with Computer Sciences Corporation and with Veridian/MRJ, and a Staff Scientist with the Research Institute for Advanced Computer Science (RIACS), all at NASA Ames. He is the Group Lead of the Algorithms, Tools,

and Architectures (ATA) Group that performs research in computer science technology for high-performance scientifi c computing. The ATA Group is part of the NASA Advanced Supercomputing (NAS) Division of NASA Ames.

Dr. Biswas has published over 85 technical papers in journals and major conferences, and has given several invited talks at home and abroad. He received the Best Paper Award for signifi cant research contributions at Supercomputing'99, and the Best Student Paper Award at Supercomputing-2000. He was awarded the NASA Contractor Council Excellence Award in 1993 for his work on developing an automatic mesh adaptation procedure for three-dimensional unstructured meshes for problems in computational fluid dynamics. He has guest-edited special issues of Parallel Computing, JPDC, and Applied Numerical Mathematics. His current research interests are in dynamic load balancing for NUMA and multithreaded architectures, analyzing and improving single-processor cache performance for irregular applications, scheduling strategies for heterogeneous distributed multi-resource servers in NASA's Information Power Grid (IPG), mesh adaptation for mixed-element unstructured grids, resource management for mobile computing, and the scalability and latency analysis of key NASA algorithms and applications. He is a member of ACM and the IEEE.

Contact Information for Each Author

SAJAL K. DAS
Department of Computer Science & Engineering
The University of Texas at Arlington
P.O. Box 19015
Arlington, TX 76019
E-mail: das@cse.uta.edu
Phone: 817-272-7405
Fax: 817-272-3784

DANIEL J. HARVEY
Department of Computer Science & Engineering
The University of Texas at Arlington
P.O. Box 19015
Arlington, TX 76019
E-mail: harvey@cse.uta.edu
Phone: 817-272-7405
Fax: 817-272-3784

RUPAK BISWAS (**Corresponding Author**)
NASA Ames Research Center
Mail Stop T27A-1
Moffett Field, CA 94035
E-mail: rbiswas@nas.nasa.gov
Phone: 650-604-4411
Fax: 650-604-3957