Parallel Processing of Adaptive Meshes with Load Balancing

Sajal K. Das and Daniel J. Harvey
Department of Computer Sciences
University of North Texas
P.O. Box 311366
Denton, TX 76203-1366
{das,harvey}@cs.unt.edu

Rupak Biswas
MRJ Technology Solutions
NASA Ames Research Center
Mail Stop T27A-1
Moffett Field, CA 94035-1000
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 the sense 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 performance of the SBN-based load balancer is comparable to results achieved under PLUM.

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 processing many scientific applications in parallel. These applications 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 [12, 15]. 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 the 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.

In 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 [10, 11, 12, 16, 21, 23, 26]. 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 execution cost. In a multilevel approach, the grid graph is first contracted to a small number of vertices and edges, and the coarsened graph is then partitioned and successively refined using the Kernighan-Lin replacement algorithm [17]. However, other partitioning methods have also been developed, and an excellent survey is provided in [1].

Although several dynamic load balancers have been proposed for multiprocessor platforms [3, 4, 6, 13, 18, 24, 25], 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 [8]. Our earlier experiments with synthetic loads have demonstrated that [6] an SBN-based load balancing solution achieves superior performance when compared to other popular techniques such as Random, Gradient, Receiver Initiated, Sender Initiated, and Adaptive Contracting.

The SBN-based load balancer proposed in this paper and targeted for adaptive meshes, can be classified as:

- **Adaptive:** Processing automatically adjusts to the number of jobs that are queued.
- **Decentralized:** Responsibility for load balancing is shared by all the nodes of the system. Any node can initiate load balancing activity.
- **Stable:** Excessive load-balancing traffic does not burden the network, especially under extremely light or heavy system loads.
- **Effective:** System performance does not degrade because of load balancing activities.

Recently, experiments that measure the effectiveness of load balancing adaptive meshes have been conducted using an automatic portable environment, called PLUM [19], developed at NASA Ames Research Center. PLUM uses a novel strategy for load balancing which consists of two separate phases: repartitioning and remapping. After each mesh adaptation step, the computational grid is globally repartitioned if the workload distribution is unacceptable. The new partitions are then reassigned among the processors in such a way that minimizes the cost of data movement. If the remapping cost is compensated by the computational gain that would be achieved with balanced partitions, then only is the necessary data appropriately redistributed. Otherwise, the new partitioning is discarded. Note that data is not physically migrated unless the cost estimates indicate that doing so is beneficial. the salient differences between the SBN-based load balancer and PLUM are given later.

We have conducted extensive experiments on an IBM SP2 to compare the performance of the SBN-based load balancer to the results obtained under PLUM in [2]. The results demonstrate that our SBN-based algorithm achieves excellent load balance, and that the redistribution cost is significantly lower than those obtained under PLUM by using two state-of-the-art partitioners, PMeTiS and DMeTiS (cf. Sec. 4.2). However, the edge cut percentages are higher than those for PMeTiS, indicating that the SBN strategy reduces the redistribution cost at the expense of a higher communication cost. For example, with a network of 32 processors, the redistribution cost using the SBN strategy is approximately half of the cost incurred under PLUM. But the total communication under SBN is double that experienced under PLUM. In many mesh adaptation applications in which the data redistribution cost dominates the processing and communication cost, this is an acceptable trade-off. A preliminary version of this paper appeared in [7].

This paper is organized as follows. Section 2 reviews the definition and properties of symmetric broadcast networks (SBN). Section 3 describes the load balancing algorithm which incorporates a global view needed for adaptive mesh applications. This section also describes a pre-partitioner that is optionally used with the SBN approach to assign subgrids to the processors before each adaptation step. Section 4 presents an overview of PLUM and the partitioners operating under that environment that are used for our comparisons. Section 5 presents experimental results and a comparative performance analyses. Section 6 concludes the paper.

2 Symmetric Broadcast Networks

A *symmetric broadcast network* (SBN), proposed by Prasad and first presented in [8], defines a communication pattern (logical or physical) among the P processors in a multicomputer system. This communication pattern can be efficiently embedded into different parallel architectures in a topology-independent manner [5, 9]. We give a brief overview of SBN and its properties.

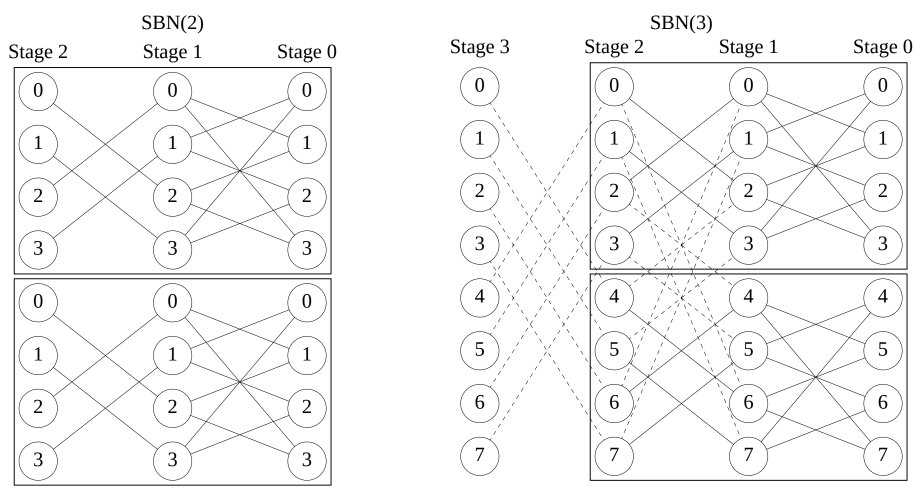

Figure 1: (a) Construction of an SBN(2) from a pair of SBN(1)s, and (b) an SBN(3) from a pair of SBN(2)s. The new connections are shown by solid lines and the original connections by dashed lines.

Definition 1 *An SBN of dimension $d \geq 0$, denoted as SBN(d), is a $(d+1)$-stage interconnection network with $P = 2^d$ processors in each stage. It is constructed recursively as follows. A single node forms the basis network SBN(0). For $d > 0$, an SBN(d) is obtained from a pair of SBN$(d-1)$s by adding a communication stage in the front with the following additional interprocessor connections:*

- *Node i in stage 0, is made adjacent to node $j = (i + P/2) \bmod P$ of stage 1; and*
- *Node j in stage 1 is made adjacent to the node in stage 2 which was the stage 0 successor of node i in SBN$(d-1)$.*

Figure 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.

By definition, each node in every stage s, for $1 \leq s \leq d-1$, of SBN(d) has exactly two successors; whereas a node in stage 0 has only one successor and a node in stage d has no successors.

The SBN(d) defines unique communication patterns (or broadcast trees) among the nodes in the network. To be more precise, for any source node or root 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 nodes appears exactly once.

Lemma 1 *Let n_x^s be a node at stage s in the broadcast tree T_x having the root node x at stage 0, where $0 \leq x < P$. Then $n_x^s = n_0^s \oplus x$, where $\oplus$ is the exclusive-OR operator, thus leading to $T_x = T_0 \oplus x$.*

In other words, all SBN communication patterns can be derived from the template tree with node 0 as the root. As an example, consider two communication patterns T_0 and T_5 in SBN(3) as shown in Figs. 2(a) and 2(b), respectively. By our convention, n_0^s denotes a node at stage s in Fig. 2(a) while n_5^s is the corresponding node in Fig. 2(b). Furthermore, $n_5^s = n_0^s \oplus 5$.

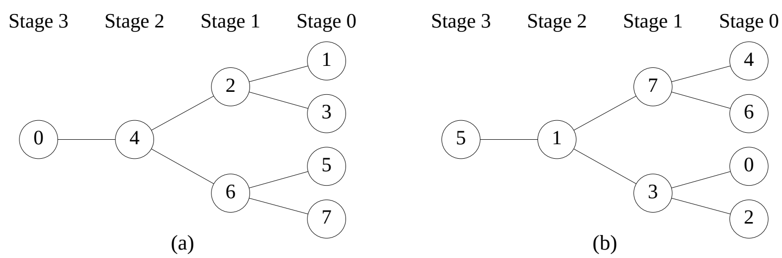

Figure 2: Examples of SBN communication patterns in SBN(3).

The predecessor and successors of each node of SBN(d) are also uniquely defined by specifying the root x and the communication stage s so that messages from x can be appropriately routed to the other nodes.

Lemma 2 *The predecessor and successors of node n_0^s can be computed as:*

Predecessor $= (n_0^s - 2^{d-s}) \vee 2^{d-s+1}$, *where $\vee$ is the inclusive-OR operator.*

Successor_1 $= n_0^s + 2^{d-s-1}$, *for $0 \leq s < d$.*

Successor_2 $= n_0^s - 2^{d-s-1}$, *for $1 \leq s < d$.*

A unique SBN communication pattern corresponding to the root node 0 can also be obtained following the one-dimensional array representation of a full binary tree.

Lemma 3 *The predecessor and successors of node n_0^s are then defined as follows:*

Predecessor $= \lfloor n_0^s \ / \ 2 \rfloor, \quad if \ s > 0.$

$$Successor\_1 = \begin{cases} 1, & if \ s = n_0^s = 0 \\ 2 \times n_0^s, & if \ 1 \leq s < d. \end{cases}$$

Successor_2 $= 2 \times n_0^s + 1, \quad if \ \ 1 \leq s < d.$

It is worth mentioning the versatility of SBNs as a topology. As shown in [5], SBN(d) can be efficiently embedded on to the d-dimensional hypercube with the help of a *modified binomial spanning tree* (MBST), in which two binomial trees are connected back to back. Figure 3 shows such a communication pattern MBST in the network SBN(4), where the broadcast messages originate

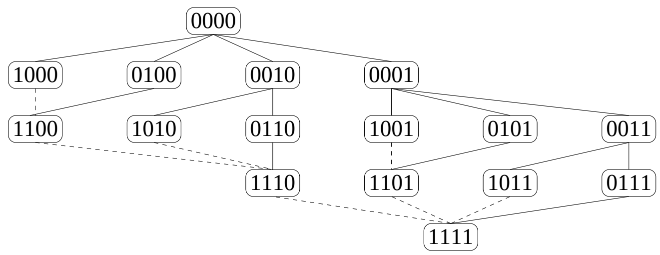

Figure 3: Binomial spanning tree used for embedding SBN(4) onto a hypercube.

from node 0 and terminate at node 15. The solid lines in this figure represent the actual SBN pattern, whereas the dashed lines are used to gather messages at the single destination node.

Clearly, the MBST is particularly suitable for adapting the SBN algorithm to the hypercube architecture. It ensures that all successor and predecessor nodes at any communication stage are adjacent nodes in the hypercube. Also, every originating node has a unique destination node. If the nodes are numbered using a binary string of d bits, the number of predecessors for a node is $\max(b, 1)$ where b is the number of consecutive leftmost 1-bits in tht node's binary address. We are currently investigating how to efficiently embed SBNs into other topologies such as meshes and toruses.

3 SBN-Based Load Balancer

The proposed load balancer adapts is behavior according to the system load. It takes into account a global view of the system to make it effective for adaptive mesh applications. Under heavy (light) load, the balancing activity is primarily initiated by processors that are lightly (heavily) loaded. This activity is controlled by two thresholds, `MinTh` and `MaxTh`, for the system load levels. Our load balancer processes two types of messages: (i) load balance messages and (ii) job distribution messages.

The first message type is broadcast when a processor p determines that its weighted queue length `QWgt`(p) <`MinTh`, the minimum threshold. In our experiments, `MinTh` was set to reflect a 0.1 second processing load. A load balancing message will also be broadcast if `QWgt`(p) >`MaxTh`, a maximum threshold, or if distributing the excess jobs will result in other processors exceeding `MaxTh`. As the load balancing message passes from one node to another, values for the global weighted queue length (`GWLen`) and the average weighted system load level (`WSysLL`) are computed.

The second message type that is broadcast is used to distribute jobs when `QWgt`(p) >`MaxTh`. Distribution messages are also used to complete the load balancing process. After the `WSysLL` value is calculated, a distribution message is broadcast through the network so that jobs are routed to lightly loaded processors and the system control variables (`MinTh`, `MaxTh`, and `WSysLL`) can be updated throughout the network. As a result, the workload at all nodes is balanced. Assuming that communication from a node to its neighbors requires constant time, it has been shown [8] that a single load balancing operation requires $O(\log P)$ time in the SBN, and that simultaneous processing of multiple balancing operations requires $O(\log^2 P)$ time in the worst case. To reduce message traffic, a node does not initiate additional load balancing activity until all the previous messages that have passed through the node have been completely processed.

The performance of the SBN-based load balancer is influenced by the choice of values for `MinTh` and `MaxTh`. For example, if `MinTh` is too small, a node 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 network, very little load balancing activity should be required.

Note that it is possible to encounter a situation when there are so many jobs in the system that at least one node 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 node at the last stage determines that its `MaxTh` has exceeded, it triggers a load balance message instead of distributing the excess load. As a result, `WSysLL` and `MaxTh` are recomputed throughout the network.

Figure 4 presents a pseudo code overview of the SBN-based load balancing algorithm. The procedures, *GetDistribute* and *GetBalance*, are used to respectively process distribution messages and balance messages that are received. Similarly, the two other procedures, *Distribute* and *Balance*, respectively route distribution and balance messages to the successor nodes in the SBN. Details of these procedures can be found in [5, 6]. Figure 4 also presents a pseudo code for the *UpdateLoad* procedure which is called for updating system thresholds. Note here that the `MaxTh` value increases rapidly as `SysLL` increases, and `MinTh` is never larger than a user-supplied parameter, `Const`. An empirical study [6] shows that the best results are obtained with `Const` set to reflect 0.1 seconds of processing load. Setting `Const` to a small value is also consistent with the analysis of gradient-based load balancing algorithms [22].

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

Procedure *Main Line Processing*

Repeat forever
    Call *GetBalance* to process balance messages
    Call *GetDistribute* to process job distributions
    If ($\texttt{QWgt}(p) > \texttt{MaxTh}$)
        Call *Distribute* to route excess jobs
    If ($\texttt{QWgt}(p) < \texttt{MinTh}$)
        Call *Balance* to initiate load balancing
        Call *UpdateLoad*(`GWLen`) to set `WSysLL`
    Normal Processing
End Repeat

Procedure *UpdateLoad*(`GWLen`)
    $\texttt{SysLL} = \lceil \texttt{GWLen}/P \rceil$
    $\texttt{MaxTh} = \texttt{WSysLL} + 2^{\lfloor \texttt{WSysLL/Const} \rfloor}$
    $\texttt{MinTh} = \max(0, \texttt{WSysLL} - \texttt{Const})$
    If ($\texttt{WSysLL} > \texttt{Const}$)
        $\texttt{MinTh} = \texttt{Const}$
Return

Figure 4: Pseudo code for SBN-based load balancer.

3.1 Weighted Queue Length

The queue length of a processor p by itself is not an accurate estimate of the amount of time required to completely service the vertices in its local queue, particularly in applications where the meash is adapted. To achieve a better balance, we must take into account the system variables like computation, communication, and redistribution costs, that affect the processing of a local queue. Therefore, we define a new metric called weighted queue length, $\texttt{QWgt}(p)$. The threshold values, $\texttt{MinTh}$ and $\texttt{MaxTh}$, are set based on the $\texttt{QWgt}(p)$ values.

Let Wgt^v be the computational cost to process a vertex v, $Comm_p^v$ be the communication cost to interact with the vertices adjacent to v but whose data sets are not local to p, and $Remap_p^v$ be the redistribution cost to copy the data set for v to p from another processor. These factors vary greatly from one vertex to another in a given mesh. $\texttt{QWgt}(p)$ is defined as:

$$\texttt{QWgt}(p) = \sum_{v=1}^{\texttt{QLen}(p)} (Wgt^v + Comm_p^v + Remap_p^v).$$

Clearly, if the data set for v is already assigned to p, no redistribution cost is incurred, i.e., $Remap_p^v = 0$. Similarly, if the data sets of all the vertices adjacent to v are already assigned to p, there is no communication cost, i.e., $Comm_p^v = 0$.

3.2 Weighted System Load

The weighted system load level, $\texttt{WSysLL}$, is computed as:

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

where P is the total number of processors used.

Assuming that the load is perfectly balanced among the processors, the metric $\texttt{WSysLL}$ estimates the time required to process the mesh and reflects the processing, communication, and redistribution costs in the current mesh-to-processor assignment. Hence, a global view of the system is captured.

3.3 Prioritized Vertex Selection

When selecting vertices to be processed, the SBN-based load balancer takes advantage of the underlying structure of the adaptive mesh and defers execution of boundary which could otherwise be migrated for more efficient execution. Thus, the selection of which queued vertex is to be executed next is made such that the overall cut size 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 $(Comm_p^v + Remap_p^v)/Wgt^v$. Therefore, vertices with no communication and redistribution costs are executed first, followed by vertices with low communication or redistribution overhead relative to their computational weight. Conceptually, internal vertices are processed before those on partition boundaries.

3.4 Differential Edge Cut

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

$$\Delta\texttt{Cut} = Remap_q^v - Remap_p^v + (Comm_q^v - Comm_p^v).$$

If $\Delta\texttt{Cut} > 0$, it is normalized as $\Delta\texttt{Cut}/Wgt^v$.

The parameters $Remap_p^v$ and $Remap_q^v$ will either be 0 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 $Remap_p^v = Remap_q^v = 8$. On the other hand, if the data for v resides on $p = 3$, then $Remap_p^v = 0$ but $Remap_q^v = 8$.

A positive $\Delta\texttt{Cut}$ indicates that an increase in communication and redistribution costs will result if v is migrated from p to q. Therefore, the formula favors migrating vertices with the smallest increase in communication cost per unit computational weight. In contrast, negative $\Delta\texttt{Cut}$ values indicate a reduction in communication and redistribution costs, hence favoring the migration of vertices with the largest absolute reduction in communication and redistribution costs.

Once $\Delta\texttt{Cut}$ is calculated for all the randomly chosen vertices, the vertex `MinV` with the smallest value of $\Delta\texttt{Cut}$ is chosen for migration. Next, following a breadth-first search, the SBN balancer selects the vertices adjacent to `MinV` that are also queued locally for processing at 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 vertices still need to be migrated, another subset of vertices are randomly chosen and the procedure is repeated. This migration policy strives to maintain or improve the cut size during the execution of the load balancing algorithm.

3.5 Data Redistribution Policy

The redistribution of data is performed in a "lazy" manner. Namely, the data set for a given vertex v in a processor p is not moved to another processor q until the latter is about to execute v. Furthermore, the data sets of all vertices adjacent to v that are also assigned to q are migrated as well. This policy greatly reduces both the redistribution and communication costs by avoiding multiple migrations of data sets and having resident all adjacent vertices that are assigned to processor q while v is being processed.

We implement data migration by broadcasting a job migration 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. Therefore, all processors consistently maintain the location of all data sets.

This policy is expected to maximize the number of adjacent vertices that are local when a grid point is processed. Hence, by considering the underlying mesh structure, the communication overhead is reduced.

3.6 An Illustrative Example

Figure 5 illustrates the SBN-based load balancer described above. 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 within parentheses. For instance, vertex 9 has a processing cost of 1 and a redistribution cost of 2. Adjacent vertices on the diagram

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

are connected by edges which are labeled with the associated communication cost. For example, the communication cost between vertices 10 and 11 is 3, provided the data sets for the two vertices reside on different processors when either one is processed. Each of the four processors are shaded differently in the figure. For example, vertices 1, 5, 6, and 9 are assigned to $P2$. We also assume that the data for vertex 7 is resident in $P1$, the data for vertex 10 is in $P2$, while the data for vertices 9, 11, and 16 are resident in $P0$. The data sets for the remaining vertices are resident in the processor to which the vertices are assigned. Table 1 shows the Wgt^v, $Comm_p^v$, and $Remap_p^v$ values for each vertex v, under the current vertex-to-processor assignment.

Table 1: Computational, communication, and redistribution costs for each vertex, v.

v	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p	P2	P1	P3	P3	P2	P2	P3	P3	P2	P0	P3	P3	P0	P0	P0	P3
Wgt^v	1	1	2	1	1	3	4	1	1	2	3	1	1	1	2	1
$Comm_p^v$	6	2	3	0	3	6	12	0	5	0	3	3	3	5	2	3
$Remap_p^v$	0	0	0	0	0	0	7	0	2	1	1	0	0	0	0	1

Table 2 shows the $\texttt{QWgt}(p)$ values for each processor p, obtained by adding the Wgt^v, $Comm_p^v$, and $Remap_p^v$ costs of the vertices assigned to p (cf. Sec. 3.1). The weighted system load is calculated by taking the average of all the $\texttt{QWgt}(p)$ values: $\texttt{WSysLL} = \lceil 94/4 \rceil = 24$ (cf. Sec. 3.2).

Table 2: Weighted queue length for each processor, p.

p	Wgt^v	$Comm_p^v$	$Remap_p^v$	$\texttt{QWgt}(p)$
P0	6	10	1	17
P1	1	2	0	3
P2	6	20	2	28
P3	13	24	9	46

Assume that $\texttt{MinTh} = 10$ reflecting that 100 jobs are executed per second, with each job requiring 0.1 second of processing. Clearly, $P1$ is 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

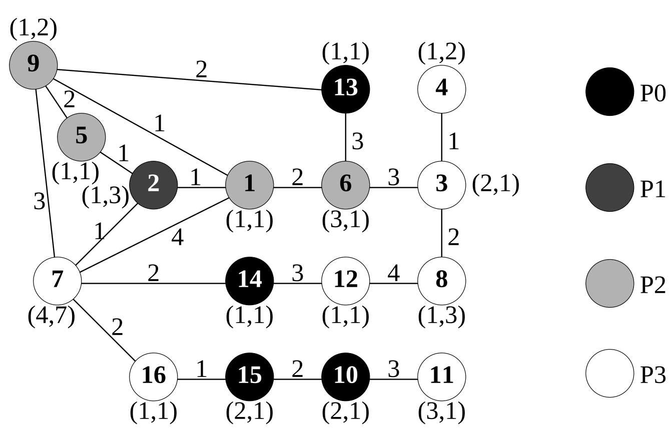

Figure 5: An example to illustrate the SBN-based load balancer.

which vertices to migrate to $P1$ so that their loads will be equidistributed. Let us step through the process of selecting the initial vertex to migrate. The decision is based on the $\Delta\mathtt{Cut}$ values of the vertices currently assigned to $P3$, and shown in Table 3 (cf. Sec. 3.4). Vertex 7 is found to be optimal for migration from $P3$ to $P1$, yielding $\mathtt{QWgt}(P3) = 23$ and $\mathtt{QWgt}(P1) = 18$. The new value of $\mathtt{WSysLL} = \lceil 86/4 \rceil = 22$ reflects an improvement in the total system load. For this example, additional migration is not necessary.

Table 3: Differential edge cut for each vertex in $P3$ if migrated to $P1$.

v	3	4	7	8	11	12	16
$Remap^v_{\mathrm{P1}}$	1	2	0	3	1	1	1
$Remap^v_{\mathrm{P3}}$	0	0	7	0	1	0	1
$Comm^v_{\mathrm{P1}}$	6	1	11	6	3	7	1
$Comm^v_{\mathrm{P3}}$	3	0	12	0	3	3	3
$\Delta\mathtt{Cut}$	4	3	-8	9	0	5	-2

3.7 SBN Pre-partitioner

The SBN load-balancing algorithms are designed to run dynamically without the need for a separate partitioning process. This is a significant advantage over the existing approaches taken by most parallel adaptive mesh applications implemented to date. Those methods must suspend processing when processor loads become imbalanced. During the suspension, mesh vertices are reassigned and the corresponding data sets are remapped. Another advantage of the SBN approach is that it allows overlapping of the computational, communication, and redistribution phases which would lead to further reductions in the overall execution time. Existing methods which separately process the computational, partitioning, and remapping phases of adaptive mesh applications cannot achieve this overlap.

To test the effectiveness of the SBN approach, we have implemented a pre-partitioner which can optionally be run prior to each mesh adaptation phase. Since state-of-the-art parallel partitioners execute quickly (less than a second for a million vertices on 64 processors), we wanted to determine whether separately running a front-end partitioner would have significant benefit on the resulting communication and/or redistribution overhead. Our pre-partitioner computes a P-way partition of the mesh vertices and then uses the default partition-to-processor assignment as the starting point for subsequent SBN-based load balancing. This pre-partitioner is non-standard in the sense that it partitions the mesh based on $\mathtt{QWgt}(p)$ values, which take into consideration all three factors of computation, communication, and redistribution. In addition, the data associated with each mesh vertex is not migrated after the partitioning process is completed. The actual data movement takes place during mesh adaptation as vertices are processed with SBN load balancing in effect.

Since the initial partition-to-processor assignment represents only a starting point, the partitioning capabilities inherent in the SBN-based balancing algorithm could significantly alter the processor assignments as the mesh is subsequently processed. These adjustments occur dynamically as individual processors become overloaded or underloaded. Our experiments demonstrate that the SBN pre-partitioner produces lower communication costs and higher data remapping costs than when the pre-partitioning option is not used (cf. Tables 4 and 5). Although the SBN pre-partitioner may be of limited value for those adaptive mesh applications where remapping costs dominate communication costs, it is sketched below for the sake of completeness.

The SBN pre-partitioner is a diffusive algorithm which uses the original mesh as a starting point and improves balancing by reassigning vertices among processors. It computes a P-way partition such that $\texttt{QWgt}(p)$ is approximately equal for all the processors and $\texttt{WSysLL}$ is minimized. Since $\texttt{QWgt}(p)$ is the sum of the computational, communication, and redistribution costs, this is a somewhat stronger requirement than that considered in some other approaches [12, 15] where the mesh is partitioned to equalize the total computational cost while minimizing the total number of cut edges. Therefore, if only a few processors incurred most of the communication overhead, significant idle time could result during processing.

The pre-partitioner differs from the partitioning capabilities inherent in the SBN load balancer in that multiple iterations are executed to find an optimal P-way partition. Here, an *iteration* is defined as a sequence of vertex reassigments from one processor to another. During an iteration, each vertex is allowed to 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 and belonging to q. First, the $\Delta\texttt{Cut}$ value is computed for all adjacent vertices assigned to processors other than q. The non-local adjacent vertex $\texttt{MinV}$, one with the smallest $\Delta\texttt{Cut}$, is then added to the set of vertices assigned to q. In addition, a breadth-first search is performed on the vertices adjacent to $\texttt{MinV}$ that are not assigned to q but to p where $\texttt{QWgt}(p) > \texttt{WSysLL}$. These vertices are also assigned to q.

At the end of an iteration when all vertices have been considered, the partitioner computes the load imbalance factor $\texttt{QWgt}(r)/\texttt{WSysLL}$, for the processor r with the largest value of $\texttt{QWgt}(r)$. If the load imbalance factor is greater than a specified threshold (1.75 in our experiments), the Kernighan-Lin refinement procedure [17] is invoked to further reduce $\texttt{WSysLL}$.

Initially, the pre-partitioner is set to execute a fixed number of iterations, $\texttt{ItNum}$. In our experiments, $\texttt{ItNum} = 4$ produced the best results. After each iteration, the total number of iterations is increased by $\texttt{ItNum}$ if a new minimum $\texttt{WSysLL}$ is achieved. The algorithm terminates when all iterations are completed.

4 The PLUM Environment

We experimentally compare the performance of our SBN-based load balancer with PLUM [19], a portable and parallel load balancing framework for adaptive unstructured grids. For the sake of completeness, the features of PLUM are summarized in this section.

Figure 6 provides an overview of PLUM. After an initial partitioning, a `Solver` executes several iterations of the application. When the grid-to-processor mapping becomes unbalanced due to mesh adaptation, PLUM gains control to determine if the workload among the processors has become unbalanced and to take appropriate action if needed. Mesh repartitioning and processor reassignment are then performed. If the estimated remapping cost exceeds the expected computational gain to be achieved, execution continues without remapping. Otherwise, the grid is remapped among the processors before the computation is resumed.

The PLUM load balancer features (i) repeated use of the initial (mesh) dual graph during the course of an adaptive computation, (ii) parallel mesh repartitioning, (iii) an efficient remapping and data movement scheme, and (iv) accurate cost model metrics. Each of these features is discussed in the following subsections.

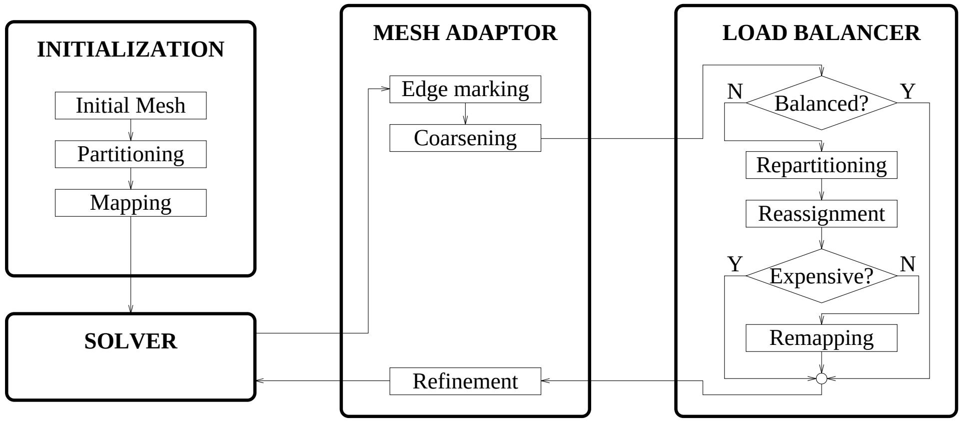

Figure 6: Overview of PLUM, a framework for globally balancing parallel adaptive computations.

4.1 Reusing the Initial Dual Graph

PLUM repeatedly utilizes the dual of the initial mesh for the purposes of load balancing. The computational weight, Wgt^v, of a vertex v in the dual graph, is set to the number of leaf elements in the corresponding refinement tree because only those elements with no children participate in the numerical computation. The redistribution cost, $Remap^v$, 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, $Comm^e$, of a dual graph edge e, is set to the number of corresponding faces in the computational mesh. The values of Wgt^v, $Remap^v$, and $Comm^e$ 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.

4.2 Parallel Mesh Repartitioning

As the computational mesh is adapted and the processor workloads need to be rebalanced, PLUM can use any general-purpose partitioner. In [2], two state-of-the-art partitioners, PMeTiS [16] and DMeTiS [23], 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 partitioner that makes no assumptions on how the mesh is initially distributed.

Both PMeTiS and DMeTiS are k-way multilevel algorithms that operate in three phases: (i) in the initial coarsening phase, the original mesh, M_0, is reduced by collapsing adjacent vertices or edges through a series of smaller and smaller meshes to M_k, such that M_k has a sufficiently small number of vertices; (ii) in the partitioning phase, the mesh workload is balanced among the processors and the edge cut size is minimized; and (iii) in the projection phase, the partitioned mesh M_k is gradually restored to its original size M_0.

DMeTiS and PMeTiS differ mainly in how they perform the partitioning phase. DMeTiS uses a directed 2-norm minimization algorithm described in [14], which provides a global picture of the existing mesh that directs the reassignment of vertices. Vertices in heavily-loaded partitions that are adjacent to neighbors in more lightly-loaded partitions are randomly visited. The diffusion algorithm 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 either the partition is balanced or no further progress towards balance can be made. If a balanced partitioning cannot be achieved at the coarsest level of the mesh, the graph is projected to the next finer level and the partitioning process repeats. 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.

During the projection phase, both PMeTiS and DMeTiS project the graph M_k back through the intermediate meshes to M_0. After each projection, a greedy refinement is performed using a variant of the Kernighan-Lin replacement algorithm [17]. This process tends to reduce the cut size further since subsequent projections allow more degrees of freedom.

4.3 Efficient Processor Remapping and Data Movement

The goal of processor reassignment is to find a mapping between partitions and processors that minimize the cost of redistribution. The first step in this direction is to compute a similarity matrix, S, where the entry S_{ij} is the sum of $Remap^v$ values of all the dual graph vertices in the new partition j that already reside in processor i. Figure 7 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. 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.

	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

Figure 7: A similarity matrix after processor reassignment.

4.4 Accurate Cost Model Metrics

Predicting the expected redistribution cost is difficult because of the large number and complexity of the costs involved. For example, the cost includes the overhead for rebuilding internal data structures and updating shared boundary information. Furthermore, redistribution costs depend on the architecture and on the many-to-many communication patterns used by the remapper. In PLUM, the total redistribution cost is estimated by individually estimating the cost of three major steps: (i) stripping data objects from a partition and placing them into a communication buffer, (ii) performing bulk transfers of information to reduce message latency, and (iii) integrating the received data into each partition. In [2], the equation $\gamma \times \texttt{MaxSR} + O$ is used to model the total cost. Here, γ represents the total cost of computation and communication 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 constant overheads such as boundary processing, data compaction, communication latency, barrier synchronization, and so on. A more accurate definition of the metric `MaxSR` is given in Sec. 5. A least squares fit has been used to approximate γ and O for various architectures, while `MaxSR` can be accurately computed from the similarity matrix.

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, control returns to the `Solver` to continue processing the unbalanced mesh.

4.5 PLUM Vs. SBN-based Load Balancer

In the following, let us point out the salient differences between the SBN-based load balancer and PLUM:

- Processing is temporarily halted under PLUM while the load is being balanced. During the suspension, a new partitioning is generated and data is redistributed among the nodes of the network. The SBN approach, on the other hand, allows processing to continue while the load is dynamically balanced. This feature also allows for the possibility of utilizing latency-tolerant techniques to hide the communication and redistribution costs during processing.
- Under PLUM, 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 that is to be achieved by a load balancing operation, processing continues using the original mesh assignment. This could result in unnecessary idle time. In contrast, the SBN approach will always result in improved load balance among the processors.
- PLUM redistributes all necessary data to the appropriate nodes immediately before processing continues. SBN, in contrast, distributes work in a "lazy" manner. In other words, 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.

5 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 adaptive calculations.

5.1 Computational Mesh Workload

The computational mesh used for the experiments 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. 8. 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. The weighted sum of vertices increased from 50,000 to 1,833,730 over the nine levels of adaptation. This test case was chosen so that the results could be compared with those compiled in [2] under the PLUM environment.

5.2 Performance Metrics

The following metrics are 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.

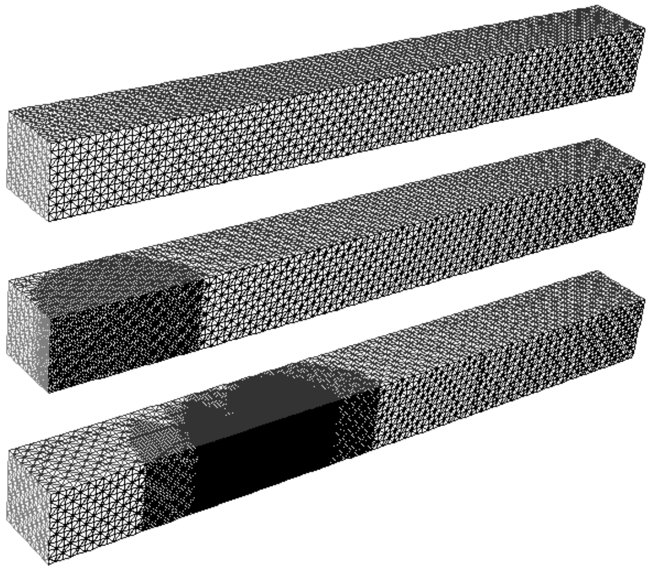

Figure 8: Initial and adapted meshes (after levels 1 and 5) for the simulated unsteady experiment.

- **Maximum redistribution cost:** The goal 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 in the following metric:

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

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

- **Load imbalance factor:** This metric 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.

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

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

  where $Comm^e$ is the weight of edge e in the adaptive mesh. The Cut% value should be as small as possible.

  Pre-Part Cut% in Table 4 initially projects the mesh edge cut before running the SBN pre-partitioner.

`Pre-Exec Cut%` computes the mesh edge cut immediately before processing a mesh adaptation level.

`Post-Exec Cut%` is the actual cut realized after processing a given adaptation level.

5.3 Summary of Results

Table 4 presents the results of processing the adaptive mesh with the SBN load balancer when running the SBN pre-partitioner between mesh adaptations. Table 5 presents similar results, but the SBN pre-partitioner is not invoked between adaptation phases. Tables 6 and 7 respectively chart the results achieved using the PMeTiS and DMeTiS partitioners within the PLUM environment. Note that Tables 6 and 7 do not show results corresponding to all values of $P = 2$, 4, 8, 16, and 32. We have included only those data sets that were available to us.

Overall, the SBN-based approach achieves excellent load balance whether the pre-partitioner is active or not. As shown in Tables 4 and 5, $\texttt{LoadImb} = 1.02$ for $P = 32$. When $P \leq 8$, an ideal load imbalance factor of 1.00 is achieved for most of the adaptation levels. In contrast, this factor is respectively 1.04 and 1.59 using PMeTiS and DMeTiS under the PLUM environment (cf. Tables 6 and 7).

The `MaxSR` metric indicates the amount of redistribution cost incurred while processing the adaptive mesh. The SBN "lazy" approach to migration of vertex data sets produces significantly lower values than those achieved by PMeTiS or DMeTiS under PLUM. For example, for $P = 32$, Table 5 shows $\texttt{MaxSR} = 28,031$, which is significantly less than the corresponding value in Table 6 ($\texttt{MaxSR} = 63,270$) and in Table 7 ($\texttt{MaxSR} = 62,542$). Additionally, when the SBN balancer is used with the partitioner, a higher `MaxSR` value results when $P \geq 8$ than when the partitioner is not active (cf. Table 5). This indicates a trade-off of lower redistribution cost for higher cut percentage. The pre-partitioner will allow an increase in the redistribution cost if it is compensated by a larger improvement in the communication cost. Comparing `Pre-Part Cut%` and `Pre-Exec Cut%` from Table 4, we also conclude that `Cut%` degrades as the pre-partitioner executes. This result is consistent with the observations drawn from the PLUM experiments.

Table 4 shows an SBN cut percentage that is more than double compared to those reported by PMeTiS (21.29% vs. 10.94% for $P = 32$). This difference in the cut percentage is significantly lower when compared to the results obtained with DMeTiS (21.29% vs. 20.22%). These results could reflect the effectiveness of the partitioners being used rather than whether the SBN balancer will always produce higher communication costs. For example, in Table 4, the cut percentage often decreases during execution of the SBN balancer. This phenomenon did not occur while experimenting with PMeTiS or DMeTiS under PLUM.

Note that the cut percentage is about 1.3 times higher when the SBN pre-partitioner is not active (cf. Tables 4 and 5). 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.

In conclusion, these experimental results demonstrate that the proposed SBN-based dynamic load balancer is effective in processing adaptive mesh applications, thus providing a global view across processors. In many mesh applications in which cost of data redistribution dominates the cost of communication and processing, the SBN balancer would be preferred.

Table 4: Mesh adaptation results using SBN load balancer with pre-partitioning.

Adaptation Level	Pre-Part Cut%	Pre-Exec Cut%	Post-Exec Cut%	MaxSR	LoadImb
$\boxed{P=2}$					
1	0.09	1.76	0.95	9,606	1.00
2	1.21	2.82	1.60	41,926	1.00
3	0.59	3.36	2.60	178,631	1.00
4	3.28	4.00	2.31	118,679	1.00
5	2.94	3.02	2.39	112,437	1.00
6	4.36	3.88	2.93	87,517	1.00
7	2.76	2.53	1.78	75,925	1.00
8	0.51	3.14	2.08	223,160	1.00
9	2.55	2.87	2.18	103,772	1.00
Average	2.03	3.04	2.09	105,739	1.00
$\boxed{P=4}$					
1	2.26	3.67	2.58	6,937	1.00
2	3.37	4.11	3.13	24,382	1.00
3	3.60	6.03	4.96	81,348	1.00
4	5.73	5.51	4.56	85,345	1.00
5	6.23	6.58	4.89	101,070	1.00
6	6.25	6.29	5.12	51,018	1.00
7	5.95	8.22	6.72	145,850	1.00
8	7.45	8.36	6.86	92,430	1.00
9	6.05	9.63	4.99	69,413	1.00
Average	5.21	6.49	4.87	73,088	1.00
$\boxed{P=8}$					
1	6.66	7.16	6.05	6,939	1.01
2	7.60	7.56	6.17	22,833	1.00
3	7.85	8.48	7.33	90,132	1.02
4	7.78	17.35	14.67	139,439	1.00
5	12.64	12.19	11.58	138,671	1.00
6	7.97	11.19	9.88	123,433	1.01
7	12.09	11.81	10.74	130,199	1.01
8	12.39	10.99	9.93	123,223	1.00
9	7.93	10.06	8.90	158,867	1.01
Average	9.21	10.75	9.47	103,748	1.01
$\boxed{P=16}$					
1	15.36	11.48	11.01	5,647	1.01
2	13.15	11.71	11.37	26,263	1.01
3	12.89	13.02	12.59	107,173	1.01
4	8.38	22.60	21.39	209,028	1.01
5	17.08	14.05	14.28	122,902	1.01
6	14.66	13.14	12.55	100,962	1.01
7	13.08	21.37	19.12	135,900	1.01
8	16.63	13.57	14.76	126,161	1.01
9	12.13	23.36	21.57	102,203	1.01
Average	13.71	16.03	15.41	104,027	1.01
$\boxed{P=32}$					
1	21.59	13.57	15.50	3,764	1.03
2	18.20	14.49	14.28	10,784	1.02
3	14.59	20.25	19.78	53,423	1.02
4	13.43	21.14	23.35	154,009	1.01
5	15.95	28.07	29.09	196,821	1.01
6	19.94	21.65	22.19	117,254	1.02
7	17.07	23.23	23.82	90,404	1.02
8	18.44	17.62	20.30	90,322	1.01
9	14.58	23.74	23.31	116,354	1.01
Average	17.09	20.42	21.29	92,571	1.02

Table 5: Mesh adaptation results using SBN load balancer without pre-partitioning.

Adaptation Level	Pre-Exec Cut%	Post-Exec Cut%	MaxSR	LoadImb
$\boxed{P=2}$				
1	0.09	4.64	6,974	1.00
2	3.14	6.18	30,538	1.00
3	5.36	6.08	57,724	1.00
4	3.93	3.86	20,646	1.00
5	2.91	5.32	76,893	1.00
6	2.33	4.62	103,544	1.00
7	2.23	5.86	140,904	1.00
8	2.83	6.14	153,735	1.00
9	3.10	6.89	129,374	1.00
Average	2.88	5.51	80,037	1.00
$\boxed{P=4}$				
1	2.26	8.15	4,078	1.01
2	7.22	10.01	26,187	1.00
3	9.44	11.69	64,110	1.00
4	9.16	9.48	46,406	1.00
5	6.60	11.86	149,042	1.00
6	9.83	10.89	94,269	1.00
7	6.58	8.00	50,337	1.00
8	2.79	15.31	170,408	1.00
9	11.53	11.48	85,152	1.00
Average	7.27	10.76	76,665	1.00
$\boxed{P=8}$				
1	6.66	10.77	2,518	1.01
2	13.93	14.98	11,109	1.00
3	15.11	18.16	46,088	1.00
4	14.65	15.83	53,032	1.00
5	11.09	16.48	69,583	1.00
6	11.02	15.91	85,982	1.00
7	13.75	18.13	105,946	1.00
8	12.84	19.51	28,974	1.00
9	15.34	17.35	80,477	1.00
Average	12.71	16.35	53,745	1.00
$\boxed{P=16}$				
1	15.36	20.61	1,767	1.01
2	24.82	25.56	7,259	1.00
3	24.40	27.45	36,031	1.01
4	20.60	22.77	43,943	1.01
5	16.11	24.27	71,736	1.01
6	17.83	22.28	66,211	1.01
7	19.75	25.00	55,361	1.01
8	17.83	25.30	64,796	1.01
9	17.87	21.59	74,316	1.01
Average	19.40	23.87	46,824	1.01
$\boxed{P=32}$				
1	21.59	26.74	1,184	1.01
2	30.35	32.32	4,387	1.02
3	30.06	34.04	8,445	1.02
4	27.28	31.43	41,783	1.01
5	21.35	29.40	42,843	1.01
6	24.04	29.42	42,688	1.01
7	22.35	30.45	41,347	1.02
8	20.59	30.48	37,006	1.02
9	22.19	29.43	32,594	1.02
Average	24.42	30.41	28,031	1.02

Table 6: Mesh adaptation results using PMeTiS under the PLUM environment.

Adaptation Level	Pre-Exec Cut%	Post-Exec Cut%	MaxSR	LoadImb
$\boxed{P = 16}$				
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
$\boxed{P = 32}$				
1	4.78	6.45	5,097	1.01
2	7.56	10.05	16,758	1.02
3	10.28	13.13	39,565	1.05
4	8.14	11.60	73,074	1.06
5	7.59	11.13	92,581	1.05
6	6.51	11.60	82,751	1.06
7	6.66	11.43	88,642	1.03
8	6.88	11.39	91,301	1.05
9	6.19	11.66	79,662	1.04
Average	7.18	10.94	63,270	1.04

5.4 Complexity Analysis

In this subsection, we analyze the overhead associated with the execution of the SBN-based load balancer and/or pre-partitioner while processing the computational mesh. Where possible, both analytic formulas and experimental data are presented.

The SBN pre-partitioner computes a P-way partition (where P is the number of available processors) of the adaptive mesh before each phase of execution. When $P = 32$, the partitioning time has been measured to be about 120 seconds, which is much higher than the average processing time (less than one second) required by the PMeTiS and DMeTiS partitioners. Note, however, that the SBN pre-partitioner runs sequentially and has not been optimized for peak performance. Furthermore, the SBN pre-partitioner does not use the multilevel approach used by PMeTiS and DMeTiS that results in significant speedup.

The overhead due to the SBN-based load balancer has four components: (i) the choice of the next vertex to be processed; (ii) the selection of the set of vertices to be migrated; (iii) the processing required to determine if load balancing is necessary; and (iv) the load balancing (communication) messages between processors. These components are analyzed below.

The next vertex v is selected using a priority min-queue. Let V_p be the set of vertices to be processed at a given processor p and let E_p be the set of all internal and border edges that are

Table 7: Mesh adaptation results using DMeTiS under the PLUM environment.

Adaptation Level	Pre-Exec Cut%	Post-Exec Cut%	MaxSR	LoadImb
$\boxed{P = 32}$				
1	4.65	15.70	5,047	1.88
2	19.26	20.50	17,393	2.12
3	21.14	25.26	44,413	2.12
4	17.13	28.21	99,232	1.87
5	29.08	26.46	97,280	1.68
6	25.31	24.38	86,204	1.41
7	20.55	14.17	78,312	1.11
8	10.04	13.08	72,474	1.05
9	9.41	14.18	62,522	1.05
Average	17.40	20.22	62,542	1.59

adjacent to the vertices of V_p. Heap operations like create and insert/delete-min require $O(V_p)$ and $O(\log V_p)$ time, respectively. The (non-standard) remove operation can be implemented in $O(\log V_p)$ time provided a direct pointer to the entry to be removed is also maintained. For SBN processing, however, $(Wgt^v + Comm_p^v + Remap_p^v)$ must be computed so that the value of $\mathtt{QWgt}(p)$ can be maintained and $(Comm_p^v + Remap_p^v)/Wgt^v$ is needed to correctly control the ordering of the SBN priority min-queue. Each of these calculations require $O(\delta_v)$ where δ_v is the degree of vertex, v. Therefore, the SBN priority queue (heap) creation requires $O(V_p + \sum_{v \in V_p} \delta_v) = O(V_p + E_p)$. Similarly, each heap insertion, delete-min, and remove operation completes in $O(\log V_p + \delta_v)$.

The vertex migration in SBN involves first selecting a random set, R, of vertices from the mesh. The vertex, $m \in R$, with the smallest $\Delta\mathtt{Cut}$ value is chosen for migration. Each $\Delta\mathtt{Cut}$ calculation completes in $O(\delta_r)$ where $r \in R$. Therefore, the time required to select the initial vertex for migration is

$$O(\sum_{r \in R} \delta_r) \approx O(|R| \times \delta_{avg}) \text{ where } \delta_{avg} \text{ is the average degree of a vertex in the mesh.}$$

Next, the mesh is searched in a breadth-first manner to choose an additional set, V_{mig}, of vertices for migration. In our experiments, to satisfy the requirements of a balance operation, $|V_{mig}|$ averages less than 10. Furthermore, a single breadth-first search almost always finds enough vertices to migrate. The time required to complete the breadth-first search is approximated by

$$O(|V_{mig}| + \sum_{v \in V_{mig}} \delta_v) \approx O(|V_{mig}| + |V_{mig}| \times \delta_{avg}).$$

Finally, each vertex to be migrated must be removed from the SBN priority queue. This operation is needed so that the vertices migrated will no longer be considered for local processing. Since $|V_{mig}| + 1$ removal operations from the heap are required, the complexity for this step is

$$O((|V_{mig}| + 1) \times \log V_p \; + \sum_{v \in V_{mig} \cup m} \delta_v) \approx O(|V_{mig}| \times (\log V_p + \delta_{avg})).$$

Combining the above terms and considering that $|R|$ is a constant, the total complexity for job migration is

$$O(|V_{mig}| + |V_{mig}| \log V_p + |V_{mig}| \times \delta_{avg}) = O(|V_{mig}| \times (\log V_p + \delta_{avg})).$$

Each processor must periodically check whether a load balancing operation is to be initiated or if load balancing messages from other processors are 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 significantly increasing processor idle time.

If f is the frequency of a processor checking for load balancing activity, we can expect the response time to process a load balance message to be $2/f$ on an average. Every time the SBN load balancer is invoked, balancing and distribution messages pass through $3 \log P$ stages of communication. 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 threshold `MinTh` should be set to a value such that load balancing will be triggered when $\texttt{QWgt}(p) < \lceil 6 \log P \times J_{avg}/f \rceil$ to avoid excessive idle time. In other words,

$$\texttt{MinTh} \geq \left\lceil \frac{6 \log P \times J_{avg}}{f} \right\rceil .$$

As an example, let $P = 32$. If an application checks for load balancing activity 60 times per second (i.e., $f = 60$) and $J_{avg} = 1000$ jobs are processed per second, then $\texttt{MinTh} \geq 500$.

Let us now measure the overheads due to message passing and processing according to our experiments. Table 8 gives the number of bytes that were transferred between processors during the load balancing and the job distribution phases. The number of bytes transferred is also expressed as a percentage of the available bandwidth. A wide-node SP2 has a message bandwidth of 36 megabytes/second and a message latency of 40 microseconds. Figure 9 plots this message passing overhead graphically, and demonstrates that the cost of vertex migration is significantly greater than the cost of actually balancing the system load. This is not unexpected. An extrapolation of the results using an exponential curve-fitting program indicates that normal speedup will not scale past 128 processors. The formula derived by the curve-fitting program for total overhead, T_{ovhd}, due to processing and message passing is obtained as:

$$T_{ovhd} = 12215.8776 \times P^{\log 0.5777} + 2.4870 \times P^{\log 1.8041} + 0.1023 \times P^{\log 1.5451}.$$

As expected, much of the overhead is due to the latency associated with transmitting many small messages which is represented by the dominating term, $P^{\log 1.8041}$, where P is the total number of processors used. However, since this exponent is less than one, the overhead is asymptotically sublinear in P. Future research will investigate utilizing latency-tolerant techniques to allow for bulk transfers.

Table 9 shows the fraction of time spent in the SBN load balancer compared to the execution time required to process the mesh adaptation application. The three columns correspond to three categories of load balancing activities: (i) the time needed to handle balance related messages, (ii) the time needed to migrate messages from one processor to another, and (iii) the time needed to select the next vertex to be processed. Figure 10 shows graphical plots. The results show that processing related to the selection of vertices is the most expensive phase of the SBN load balancer. However, the total time required to load balance is still relatively small compared to the time spent processing the mesh.

Table 8: Communication overhead of the SBN load balancer.

P	Balancing Messages (bytes)	Balancing Bandwidth (%)	Migration Messages (bytes)	Migration Bandwidth (%)
2	342,456	0.00	3,918,912	3.67
4	149,964	0.00	7,939,344	7.44
8	463,340	0.01	25,397,376	23.79
16	581,432	0.02	30,453,888	28.53
32	1,550,292	0.12	38,244,384	35.83

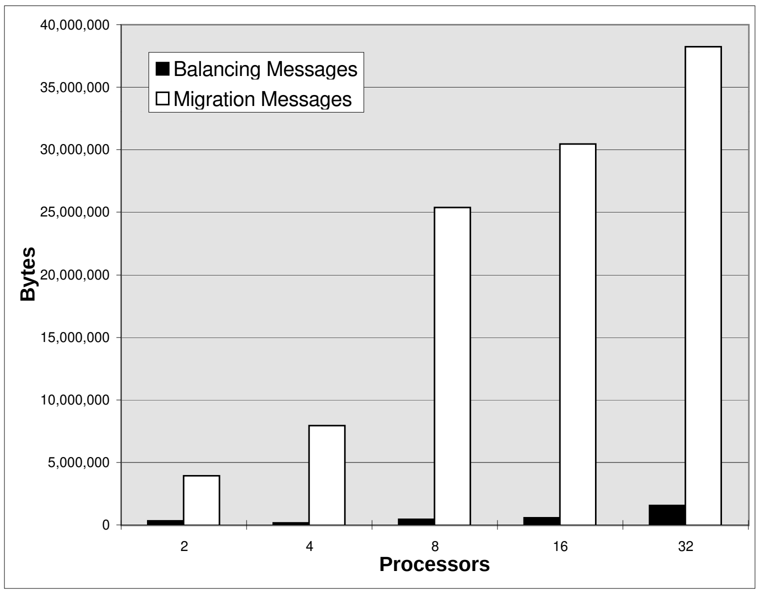

Figure 9: SBN load balancing related communication overhead.

6 Conclusions and Future Work

Our thorough experimental results show that dynamic SBN-based load balancing achieves a lower redistribution cost in processing the selected adaptive mesh application than what was achieved under the PLUM environment. However, the communication costs using SBN were significantly higher than what has been reported under PLUM. Overall, the SBN approach has proven to be a viable option in load balancing unstructured adaptive mesh applications.

We are currently experimenting on the SGI Origin2000 (a distributed shared-memory architecture) to test the consistency of our results and to implement additional performance and prepartitioning refinements to reduce the communication overhead. It would also be interesting to apply the SBN balancer to adaptive mesh applications using a heterogeneous network of computers in which P is not necessarily a power of two. Since low-cost processing power is now readily available, it would be desirable to explore the effect of adding latency-tolerant techniques to the load balancing method. This extension would allow adaptive mesh applications to be executed with good results over low-cost networks of varying topologies in contrast to homogeneous supercomputer environments such as the IBM SP2. Our research indicates that latency tolerance techniques have the potential of hiding about 75% of the communication and redistribution costs. Another area of research includes techniques to adapt the processing to situations where some of processors in the network become unavailable during a computation. Fault tolerance would allow applications

Table 9: Percentage overhead of the SBN load balancer.

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

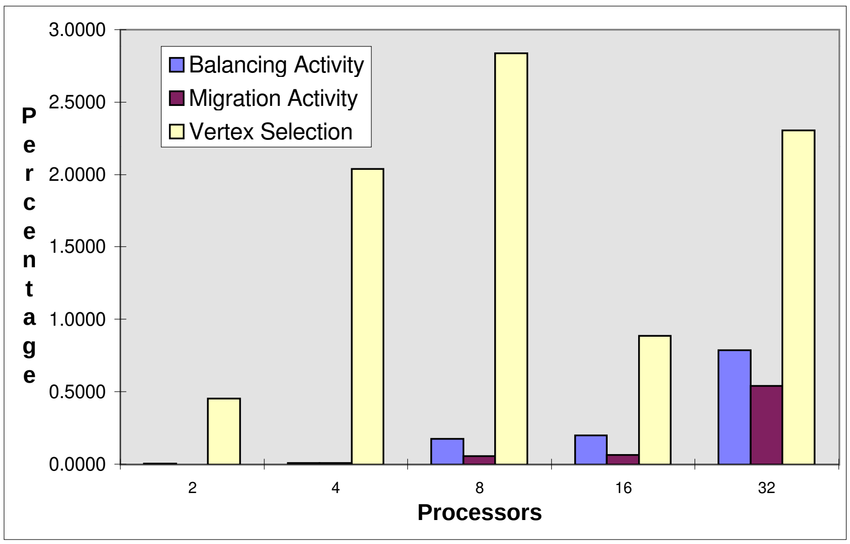

Figure 10: SBN load balancing related processing overhead.

to make use of resources that are constantly changing during execution. Finally, the techniques presented could be applied to other practical applications, such as multimedia image processing and data mining, where load balancing is an important issue.

Acknowledgements

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

References

[1] C. Alpert and A. Kahng, "Recent directions in netlist partitioning," *Integration, the VLSI Journal*, 19(1-2) (1995), pp. 1–81.

[2] R. Biswas and L. Oliker, "Experiments with repartitioning and load balancing adaptive meshes," NASA Ames Research Center, Moffett Field, CA (1997), Technical Report NAS-97-021.

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

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

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

[6] 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 International Conference on Parallel Processing* (1997), pp. 360–367.

[7] S.K. Das, D.J. Harvey, and R. Biswas, "Parallel Processing of Adaptive Meshes with Load Balancing," *Proc. 27th International Conference on Parallel Processing* (1998), to appear.

[8] S.K. Das and S.K. Prasad, "Implementing task ready queues in a multiprocessing environment," *Proc. International Conference on Parallel Computing* (1990), pp. 132–140.

[9] 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," Department of Computer Science, University of North Texas, Denton, TX (1992), Technical Report CRPDC-92-1.

[10] J. Garbers, H.J. Promel, and A. Steger, "Finding clusters in VLSI circuits," *Proc. IEEE International Conference on Computer Aided Design* (1990), pp. 520–523.

[11] L. Hagen and A. Kahng, "A new approach to effective circuit clustering," *Proc. IEEE International Conference on Computer Aided Design* (1992), pp. 422–427.

[12] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," Sandia National Laboratories, Albuquerque, NM (1993), Technical Report SABD83-1391M.

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

[14] Y. F. Hu and R. J. Blake, "An optimal dynamic load balancing algorithm," Daresbury Laboratory, Warrington, UK (1995), Technical Report DL-P-95-0-11.

[15] G. Karypis and V. Kumar, "Analysis of multilevel graph partitioning," Department of Computer Science, University of Minnesota, Minneapolis, MN (1995), Technical Report TR 95-037.

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

[17] B.W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Systems Technical Journal*, 49(2) (1970), pp. 291–307.

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

[19] L. Oliker and R. Biswas, "PLUM: Parallel load balancing for adaptive unstructured meshes," NASA Ames Research Center, Moffett Field, CA (1997), Technical Report NAS-97-020.

[20] L. Oliker, R. Biswas, and R. Strawn, "Parallel implementation of an adaptive scheme for 3D unstructured grids on the SP2," *Parallel Algorithms for Irregularly Structured Problems*, LNCS 1117, Springer-Verlag (1996), pp. 35–47.

[21] R. Ponnusamy, N. Mansour, A. Choudhary, and G.C. Fox, "Graph contraction and physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers," *Proc. 7th International Conference on Supercomputing* (1993).

[22] S. Pulidas, D. Towsley, and J.A. Stankovic, "Embedding gradient estimators in load balancing algorithms," *Proc. International Conference on Distributed Computer Systems* (1988), pp. 488–490.

[23] K. Schloegel, G. Karypis, and V. Kumar, "Multilevel diffusion schemes for repartitioning of adaptive meshes," *Journal of Parallel and Distributed Computing*, 47 (1997), pp. 109–124.

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

[25] A. Vidwans, Y. Kallinderis, and V. Venkatakrishnan, "Parallel dynamic load balancing algorithm for three-dimensional adaptive unstructured grids," *AIAA Journal*, 32 (1994), pp. 495–505.

[26] C. Walshaw, M. Cross, and M.G. Everett, "Parallel dynamic graph partitioning for adaptive unstructured meshes," *Journal of Parallel and Distributed Computing*, 47 (1997), pp. 102–108.