Dynamic Load Balancing for Adaptive Meshes using Symmetric Broadcast Networks

Sajal K. Das
Department of Computer Sciences, University of North Texas, Denton, TX 76203
das@cs.unt.edu

Daniel J. Harvey
Department of Computer Sciences, University of North Texas, Denton, TX 76203
harvey@cs.unt.edu

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

Abstract

Many scientific applications involve grids that lack a uniform underlying structure. These applications are often dynamic in the sense that the grid structure significantly changes between successive phases of execution. In parallel computing environments, mesh adaptation of grids through selective refinement/coarsening has proven to be an effective approach. However, achieving load balance while minimizing inter-processor communication and redistribution costs is a difficult problem. Traditional dynamic load balancers are mostly inadequate because they lack a global view across processors. In this paper, we compare a novel load balancer that utilizes symmetric broadcast networks (SBN) to a successful global load balancing environment (PLUM) created to handle adaptive unstructured applications. Our experimental results on the IBM SP2 demonstrate that performance of the proposed SBN load balancer is comparable to results achieved under PLUM.

1 Introduction

Mesh partitioning is a common approach to processing many scientific applications in parallel. These applications are generally modeled using a mesh (or grid) of vertices and edges. For the purpose of load balancing, each vertex is associated with a weight that indicates the amount of computational work required to process it. In a multiprocessing environment, the vertex weight contains an additional component that models the cost of redistributing the vertex from one processor to another. Similarly, each edge in the mesh has an associated weight indicating the amount of interaction between adjacent vertices.

Proceedings of the 12th ACM International Conference on Supercomputing, Melbourne, Australia, July 13–17, 1998.

In adaptive meshes, the topology of the mesh changes as the computation proceeds from one phase of the execution to the next. Traditionally, this class of problems is processed by repartitioning the graph between each phase of the execution. The goal of a partitioner is to balance the computational workload among the processors and to minimize the number of edges that are cut (and hence the overall communication cost) [11,13]. A number of partitioners designed for this purpose has been proposed in the literature [9,10,11,14,19,21,24]. These partitioners work using a multilevel approach where the graph is first contracted to a small number of vertices and edges. The coarsened graph is then partitioned and successively refined using a Kernighan-Lin replacement algorithm [15]. The multilevel approach has proven to be effective in producing good partitions at reasonable execution cost. A survey of additional partitioning methods is provided in [1].

Recently, experiments that measure the effectiveness of repartitioning adaptive meshes have been conducted using a portable environment, called PLUM [17], that was developed at NASA Ames Research Center. Figure 1 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, repartitioning and reassignment take place. 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 computation continues. Details of the parallel mesh adaptation scheme, called 3D_TAG, is described in [18]. The relative performance within PLUM of five parallel, state-of-the-art partitioning algorithms such as PMeTiS [14], UMeTiS [21], DMeTiS [21], Jostle-MD [24], and Jostle-MS [24], is reported in [2]. All of these partitioners use the multilevel approach mentioned above. UMeTiS, DMeTiS, and Jostle-MD are diffusion algorithms that modify the existing partitions; while PMeTiS and Jostle-MS create new partitions from scratch.

Although several dynamic load balancers have been pro-

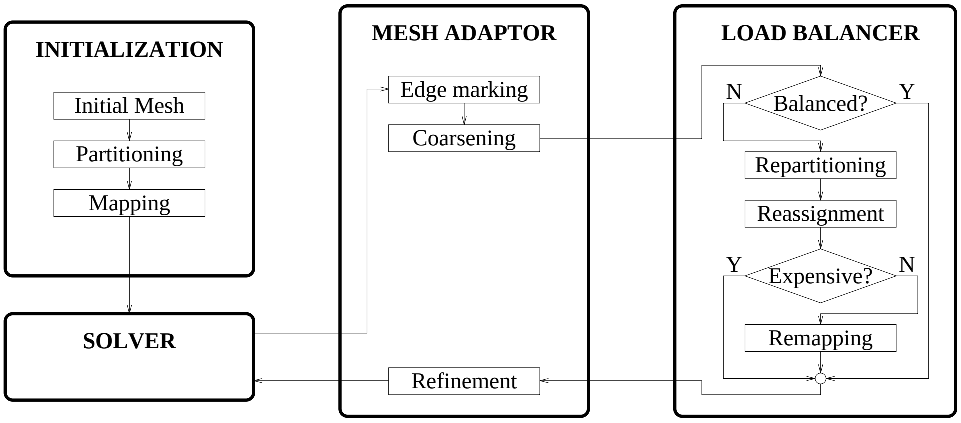

Figure 1. Overview of PLUM, a framework for globally load balancing parallel adaptive computations.

posed for multiprocessor platforms [3,4,6,12,16,22,23], most of them are inadequate for adaptive mesh applications because they lack a global view of system loads. Also, job migration in these approaches does not take into account the structure of the adaptive grid. In this paper, we overcome these deficiencies by modifying the load balancer proposed earlier by us [6]. Our load balancer makes use of a symmetric broadcast network (SBN) which is a robust and topology-independent communication pattern among processors. We also perform experiments on an IBM SP2 to compare the performance of this SBN-based load balancer to the results obtained under PLUM and reported in [2].

The experimental results demonstrate that the proposed SBN balancer achieves excellent load balance, and that the amount of redistribution cost is significantly lower than those obtained by using PMeTiS or DMeTiS under PLUM. 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.

This paper is organized as follows. Section 2 reviews the definition of SBN and a basic load balancing algorithm. Section 3 describes the modifications that were made to incorporate a global view needed for adaptive mesh applications. Section 4 describes a partitioner used with SBN to initially assign subgrids to the processors. Section 5 presents experimental results and a comparative performance analysis. Section 6 concludes the paper.

2 SBN Preliminaries

A *symmetric broadcast network* (SBN) defines a communication pattern (logical or physical) among the P processors in a multicomputer system [7,8]. This communication is topology-independent and has been efficiently embedded into different parallel architectures [5].

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 $d > 0$, SBN(d) is obtained from a pair of SBN$(d-1)$s by adding a communication stage in the front with additional interprocessor connections as follows:
  - node i, in stage 0, is connected to node $j = (i + P/2) \bmod P$ of stage 1; and
  - node j in stage 1 is connected to the node in stage 2 that was the stage 0 successor of node i in SBN$(d-1)$.

Figure 2 illustrates how an SBN(2) is constructed from two SBN(1)s. The SBN approach defines unique communication patterns among the nodes in the network. For any source node at stage 0, there are $d = \log P$ stages of communication where each node appears exactly once. Successors and predecessors of each node are uniquely defined by specifying the originating node and the communication stage. Messages originating from source nodes are appropriately routed through the SBN.

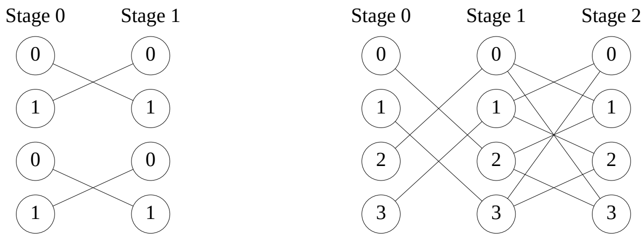

Figure 2. Construction of SBN(2) from a pair of SBN(1)s. The new connections are shown by solid lines and the original connections by dashed lines.

As an example, consider the two communication patterns for SBN(3) shown in Fig. 3. The paths in Fig. 3(a) are used to route messages originating from node 0, while those in Fig. 3(b) are for messages originating from node 5. If n_5^s denotes a node at stage s in Fig. 3(b) and n_0^s is the corresponding node in Fig. 3(a), then $n_5^s = n_0^s \oplus 5$, where $\oplus$ is the exclusive-OR operator. In general, if n_x^s is the corresponding node in the communication pattern for messages originating from a source node x, then $n_x^s = n_0^s \oplus x$. Thus, all SBN communication patterns can be derived from the template with node 0 as the root. The predecessor and two successors of node n_0^s can be computed as:

Predecessor: $(n_0^s - 2^{d-s}) \vee 2^{d-s+1}$.

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

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

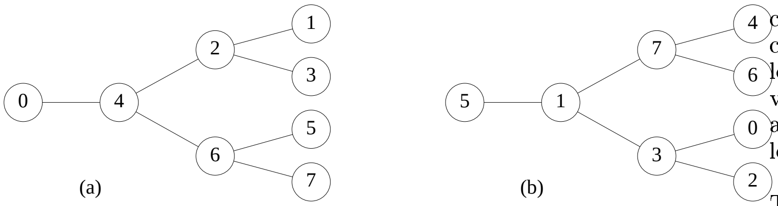

Figure 3. Examples of SBN communication patterns in SBN(3).

The processors in the SBN pattern corresponding to node 0 as the source can also be obtained following the one-dimensional array representation of a full binary tree. The predecessor and successors of node n_0^s are then defined as follows:

Predecessor: $\lfloor n_0^s/2 \rfloor$ if $s > 0$.

Successor_1: $\begin{cases} 1 + n_0^s & \text{if } s = n_0^s = 0 \\ 2 * n_0^s & \text{if } 1 \leq s < d. \end{cases}$

Successor_2: $2 * n_0^s + 1$ if $1 \leq s < d$.

To demonstrate the versatility of the SBN approach, it was adapted in [5] for use on the hypercube topology with the help of a *modified binomial spanning tree*, which is two binomial trees connected back to back. Figure 4 shows such a communication pattern for a 16-node network, SBN(4), in which the messages originate from node 0. The solid lines in this diagram represent the actual SBN pattern, whereas the dashed lines are used to gather load balancing messages at a single destination node (e.g., node 15).

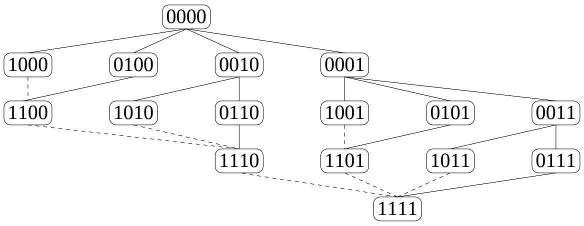

Figure 4. Binomial spanning tree used as a hypercube SBN.

The modified binomial spanning tree 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 that node's binary address.

2.1 SBN-Based Load Balancing

Let us outline the basic SBN-based load balancer which we proposed in [6]. It processes two types of messages. The first type is a load balancing message that is broadcast when a node n determines that the number of locally queued jobs (`QLen`(n)) falls below the minimum threshold (`MinTh`). A load balancing message will also be broadcast if `QLen`(n) exceeds the maximum threshold (`MaxTh`) and the excess jobs cannot be absorbed by other nodes in the network. As the load balancing message passes from one node to another, values for the global number of jobs queued (`GLen`) and the average number of jobs queued per node or the system load level (`SysLL`) are computed.

The second type of messages, for job distribution, is used to complete the load balancing process. It is broadcast through the network after the `SysLL` value is calculated. During the broadcast, jobs are routed to the lightly loaded nodes of the network. As a result, the workload at all nodes is balanced.

Assuming that communication from a node to its neighbors completes in constant time, it has been shown [7] 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 previous messages that have passed through the node have been completely processed.

Three load balancing algorithms based on the SBN concept have been implemented in [5,6]. They all use four common procedures. The first two, *GetDistribute* and *GetBalance*, are used to respectively process distribution and balance messages that are received. The other two procedures, *Distribute* and *Balance*, respectively route distribution and balance messages to the successor nodes in the SBN. Details of these procedures depend on the particular load balancing algorithm used. Figure 5 presents a pseudo code overview of the SBN-based load balancing algorithms. Although `Const` in this figure is a user-supplied parameter, experiments show that a value of 3 yields the best results [6]. Setting `Const` to a small value is also in agreement with the analysis of load balancing algorithms presented in [20].

Procedure *Main Line Processing*

Repeat forever
    Call *GetBalance* to process load balance messages
    Call *GetDistribute* to process distribution messages
    If (`QLen` $>$ `MaxTh`)
        Call *Distribute* to route excess jobs through SBN
    If (`QLen` $<$ `MinTh`)
        Call *Balance* to initiate a load balancing operation
        Call *UpdateLoad*(`GLen`) to set `SysLL`
    Normal Processing
End Repeat

Procedure *UpdateLoad*(`GLen`)
    `SysLL` $= \lceil$`GLen`$/P\rceil$
    `MaxTh` $=$ `SysLL` $+ 2^{\lfloor \texttt{SysLL/Const} \rfloor}$
    If (`SysLL` $\geq$ `Const`)
        `MinTh` $=$ `Const`
    else
        `MinTh` $=$ `SysLL` -1
Return

Figure 5. Common pseudo code for SBN-based load balancing algorithms.

3 Modified SBN Load Balancer

A series of modifications have been made to the SBN load balancer (cf. Sec. 2.1) to provide a global view of the system. These changes, which allow the SBN approach to be effective for adaptive mesh applications, are described below:

- An estimate of the amount of local processing to be performed by a processor p is no longer based on $\texttt{QLen}(p)$. Rather, the parameter $\texttt{QWgt}(p)$ is computed as an estimate of the time required to completely service the vertices queued for processing at p. The parameter takes into account all processing, communication, and redistribution costs. It is defined as:

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

  where Wgt^v is the computational cost to process vertex v, $Comm_p^v$ is the communication cost to interact with the vertices adjacent to vertex v but whose data sets are not local to processor p, and $Remap_p^v$ is the redistribution cost to copy the data set for vertex v to processor p from another processor. Clearly, if the data set for v is already assigned to p, no redistribution cost is incurred ($Remap_p^v = 0$). Similarly, if the data sets of all vertices adjacent to v are already assigned to p, the communication cost $Comm_p^v = 0$.

  Because $\texttt{QWgt}(p)$ accounts for all of the system variables that affect the processing of the local queue, the expression for $\texttt{QWgt}(p)$ gives a good estimate of the total time required to completely service the local queue.

- The formula to estimate the weighted system load level WSysLL, that is used instead of SysLL, 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.

- The choice of which queued vertex is to be executed next is made in such a way that it minimizes the overall cut size of the adaptive mesh. A priority min-queue is used to select the next vertex to be executed, where $(Comm_p^v + Remap_p^v)/Wgt^v$ is used for ordering the priority queue. Therefore, vertices without communication and redistribution costs are executed first. Next, vertices with low communication or distribution overhead relative to the computational weight of the vertex are processed. Conceptually, internal grid points are processed before those on partition boundaries.

- An optimal policy for vertex migration needs to be established when balancing the system load among processors. When vertices are being moved between processors, assume that processor p is about to reassign vertices to another processor q. The modified SBN load balancer running on p randomly picks a set of vertices from those queued locally for processing. For each selected vertex v, the differential edge cut[1] $\Delta\texttt{Cut}$ is calculated as:

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

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

  Note that $Remap_p^v$ and $Remap_q^v$ will either be 0 or equal to the redistribution cost of moving the data for vertex v from processor p to q. As an example, let $p = 3$, $q = 6$, the data for v reside on $p = 1$, and its redistribution cost be 8. In this case, $Remap_p^v = Remap_q^v = 8$. On the other hand, if the data for v resides on $p = 3$, then $Remap_p^v = 0$ and $Remap_q^v = 8$.

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

  Once $\Delta\texttt{Cut}$ is calculated for all the randomly chosen vertices, the vertex MinV with the smallest value of $\Delta\texttt{Cut}$ value is chosen for migration. Next, the SBN balancer follows in a breadth first manner the chain of vertices adjacent to MinV that are also queued locally for processing at processor 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 series of vertices are randomly picked and the procedure is repeated. This migration policy strives to maintain or improve the cut size during the execution of the load balancing algorithm.

- The redistribution of data sets is performed in a "lazy" manner. Namely, the data set for a given vertex v is not moved to processor q until it is about to execute that vertex. In this manner, the total redistribution cost is greatly reduced. Furthermore, when processor p sends the data set for vertex v to processor q, it also redistributes the data sets of all adjacent vertices assigned to p. This policy lowers the runtime communication cost because the data sets of all the adjacent vertices will reside on the processor that is processing v.

  Migration of data sets have been implemented by broadcasting a job migration message when a vertex is about to be processed and its corresponding data set is not resident. 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.

4 SBN Partitioner

An SBN partitioner is executed between mesh adaptation phases in order to optimize the cut size prior to a cycle of processing. This partitioner is a diffusion algorithm in that the existing partitioning is used as a starting point. Mesh vertices are moved between processors to balance the system load. The partitioner attempts to partition the mesh such that $\texttt{QWgt}(p)$ is approximately equal for all processors while the cut size is minimized. This is a somewhat stronger requirement than that considered in [11,13], where the mesh is partitioned so that each processor has equal computational weight while the total cut of edges is minimized. In that case, a few processors could incur most of the commu-

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

nication overhead causing some additional idle time during processing.

The SBN partitioner executes a series of iterations to balance the load. At the end of each iteration, a Kernighan-Lin refinement procedure [15] is executed to further reduce the cut size. When an iteration completes and if a new minimum cut is achieved, the total number of iterations to be executed is increased by a constant value, say 4. Upon completion of the algorithm, the partition having the minimum cut size is used as a starting point for the SBN load balancer. Note that the vertex data sets are not moved at this time, but that the migration occurs when a vertex is about to be processed.

During each iteration of the partitioner, the processor p with the minimum $\texttt{QWgt}(p)$ value is identified. A series of vertices assigned to p are then randomly selected. For each of these vertices, the $\Delta\texttt{Cut}$ value is computed for all the non-local adjacent vertices. The non-local adjacent vertex `MinV` (the one with the minimum $\Delta\texttt{Cut}$) is added to the set of vertices assigned to p. In addition, a breadth first search is performed on the vertices adjacent to `MinV` that are not assigned to p. These vertices are then migrated to p. The iteration ends when either $\texttt{QWgt}(p) > \texttt{WSysLL}$, or all vertices have been considered.

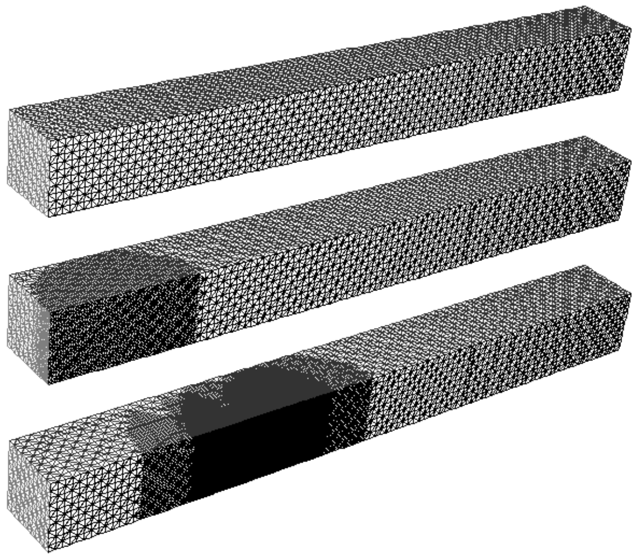

Figure 6. Initial and adapted meshes (after levels 1 and 5) for the simulated unsteady experiment.

5 Experimental Study

The modified SBN-based load balancing algorithm has been implemented using MPI and tested with actual workloads obtained from adaptive calculations on the wide-node IBM SP2 located at NASA Ames Research Center. 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 shown in Fig. 6. 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 to those compiled in [2] using the PLUM environment.

5.1 Performance Metrics

The following metrics are used to evaluate the effectiveness of the SBN load balancer when processing an unsteady adaptive mesh, where v indicates a vertex to be processed and P is the total number of processors used in the experiments.

- 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 costs:

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

  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`.

- The load imbalance factor is formulated as:

$$\texttt{LoadImb} = \max_{p\in P}\Big\{\sum_{v \text{ assigned to } p} Wgt^v\Big\} \Big/ \Big(\frac{1}{P}\sum_{p\in P}\sum_{v \text{ assigned to } p} Wgt^v\Big).$$

  The `LoadImb` value should be as close to unity as possible.

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

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

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

  `Pre-Part Cut%` in Tables 1, 3, and 4, initially projects the mesh edge cut before processing an adaptation level but after the previous adaptation level has been processed. In Table 2, the metric `Pre-Exec Cut%` is used for this purpose.

  `Post-Part Cut%` projects the mesh edge cut for processing an adaptation level after the partitioning is complete.

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

Table 1 presents the results of processing the adaptive mesh with the SBN load balancer when running the SBN partitioner between phases of mesh execution. Table 2 presents

Adaptation Level	Pre-Part Cut%	Post-Exec Cut%	MaxSR	LoadImb
$P=2$				
1	0.09	1.00	9,628	1.00
2	1.34	1.73	7,740	1.00
3	1.91	3.07	7,738	1.00
4	1.37	3.38	16,542	1.00
5	2.96	2.12	4,964	1.00
6	1.46	1.61	12,632	1.00
7	0.39	6.59	40,392	1.00
8	4.94	3.19	12,716	1.00
9	3.71	3.22	24,818	1.00
Average	2.35	3.24	15,241	1.00
$P=4$				
1	2.26	2.51	5,876	1.01
2	3.38	3.20	5,183	1.00
3	4.20	4.55	11,507	1.00
4	2.10	5.55	23,894	1.00
5	6.94	4.39	39,917	1.00
6	2.43	4.31	35,090	1.00
7	4.49	4.97	12,416	1.00
8	6.36	6.02	14,896	1.00
9	6.52	5.05	33,776	1.00
Average	4.68	4.91	20,284	1.00
$P=8$				
1	6.66	7.10	6,475	1.01
2	8.47	7.82	9,452	1.00
3	8.97	8.42	28,332	1.00
4	7.13	9.20	30,821	1.01
5	9.43	8.80	18,229	1.00
6	8.16	8.25	19,193	1.00
7	7.16	9.36	45,351	1.00
8	12.48	10.78	37,408	1.00
9	10.32	9.71	22,841	1.00
Average	9.06	9.17	24,234	1.00
$P=16$				
1	15.36	10.98	5,694	1.01
2	12.83	11.45	15,450	1.01
3	13.04	12.75	22,061	1.01
4	8.84	25.11	37,412	1.00
5	17.29	13.95	16,442	1.01
6	10.49	12.77	18,554	1.01
7	12.05	15.94	30,299	1.01
8	13.50	18.90	18,080	1.01
9	16.14	18.61	24,103	1.01
Average	13.08	16.71	20,899	1.01
$P=32$				
1	21.59	16.86	3,554	1.07
2	20.76	16.31	9,484	1.02
3	17.47	16.52	13,708	1.02
4	13.04	22.13	31,787	1.01
5	13.20	30.68	21,666	1.03
6	19.83	19.47	18,568	1.01
7	15.65	29.24	21,112	1.01
8	20.40	19.97	13,744	1.02
9	13.23	30.78	17,658	1.01
Average	16.30	24.02	16,809	1.02

Table 1. Mesh adaptation results using SBN load balancer with a partitioner.

Adaptation Level	Pre-Exec Cut%	Post-Exec Cut%	MaxSR	LoadImb
$P=2$				
1	0.09	4.71	8,712	1.00
2	3.91	7.19	14,908	1.00
3	3.64	5.84	14,800	1.00
4	5.06	4.43	3,668	1.00
5	3.98	4.26	2,456	1.00
6	1.92	5.55	44,232	1.00
7	4.39	7.60	31,800	1.00
8	5.80	5.77	11,618	1.00
9	0.91	7.98	30,614	1.00
Average	3.64	5.96	18,090	1.00
$P=4$				
1	2.26	8.24	3,931	1.01
2	8.00	9.36	17,302	1.00
3	8.80	11.93	19,696	1.00
4	9.59	10.72	31,559	1.00
5	5.94	14.51	29,396	1.00
6	8.75	12.79	21,389	1.00
7	5.51	13.74	30,232	1.00
8	13.56	13.53	22,954	1.00
9	7.70	12.16	25,345	1.00
Average	8.46	12.65	22,423	1.00
$P=8$				
1	6.66	11.14	2,412	1.01
2	14.35	15.63	10,789	1.00
3	16.80	18.34	17,347	1.00
4	11.35	14.87	32,527	1.00
5	10.13	16.07	20,753	1.00
6	13.34	17.39	25,417	1.00
7	13.28	19.16	25,508	1.00
8	15.89	17.04	29,392	1.00
9	10.95	15.94	19,026	1.00
Average	12.99	16.84	20,352	1.00
$P=16$				
1	15.36	20.60	1,775	1.01
2	24.73	25.69	4,573	1.00
3	25.07	27.97	16,195	1.01
4	19.21	22.83	19,086	1.01
5	14.65	24.41	18,080	1.01
6	18.29	24.05	19,682	1.01
7	19.68	23.89	12,703	1.01
8	17.74	22.77	15,774	1.01
9	16.44	22.56	17,370	1.01
Average	18.71	23.98	13,915	1.01
$P=32$				
1	21.59	27.26	1,351	1.06
2	31.22	32.47	2,715	1.02
3	30.60	35.37	7,932	1.01
4	28.28	31.64	12,713	1.03
5	19.13	31.37	12,318	1.01
6	20.91	32.05	9,167	1.01
7	23.59	30.17	10,698	1.01
8	22.04	29.39	10,830	1.01
9	22.74	29.83	10,223	1.01
Average	23.91	31.28	8,661	1.02

Table 2. Mesh adaptation results using SBN load balancer without a partitioner.

similar results, but the SBN partitioner is not invoked between adaptation phases. Tables 3 and 4 respectively chart the results achieved using the PMeTiS and DMeTiS partitioners within the PLUM environment. Note that Tables 3 and 4 do not show results corresponding to all values of P. We have included only the data that were available to us.

Adaptation Level	Pre-Part Cut%	Post-Part Cut%	MaxSR	LoadImb
$P = 16$				
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	5.29	8.24	114,715	1.02
$P = 32$				
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	7.30	10.85	61,221	1.04

Table 3. Mesh adaptation results using PMeTiS under the PLUM environment.

Adaptation Level	Pre-Part Cut%	Post-Part Cut%	MaxSR	LoadImb
$P = 32$				
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	20.30	62,542	1.59

Table 4. Mesh adaptation results using DMeTiS under the PLUM environment.

5.2 Summary of Results

The SBN balancer achieves excellent load balance whether the partitioner is active or not. For example, Tables 1 and 2 show that for $P = 32$, an average load imbalance factor of 1.02 is achieved. When $P \leq 8$, an ideal load imbalance factor of 1.00 is achieved for most of the adaptation levels. In contrast, the load imbalance factors using PMeTiS and DMeTiS under the PLUM environment for $P = 32$ are 1.04 and 1.59 respectively (cf. Tables 3 and 4).

The `MaxSR` metric indicates the amount of redistribution cost that is 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 1 shows $\texttt{MaxSR} = 16,809$ which is significantly less than the corresponding values in Table 3 ($\texttt{MaxSR} = 61,221$) and in Table 4 ($\texttt{MaxSR} = 62,542$). Additionally, when the SBN balancer is used with partitioning, a higher `MaxSR` value results for $P \geq 8$ than when the partitioning is not active (cf. Table 2). This indicates a tradeoff of lower redistribution cost for higher cut percentage. The partitioner will allow an increase in the redistribution cost if it is compensated by a larger improvement in the communication cost.

Table 1 shows an SBN cut percentage that is more than double compared to those reported by PMeTiS (24.02% compared to 10.85% for $P = 32$). This difference in the cut percentage is significantly lower when compared to the results obtained with DMeTiS (24.02% compared to 20.30%). 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 1, 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 at least 1.3 times higher when the SBN partitioner is not active (compare Tables 1 and 2). This demonstrates that it is useful to initially partition the mesh to compute a starting point for subsequent SBN load balancing.

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.

6 Future Work

This research could be extended by investigating the use of different partitioners in connection with the SBN approach. The partitioner described in this paper could also be refined for improved performance. We are currently experimenting on the SGI Origin2000 (a distributed shared-memory architecture) to test the consistency of the results.

It would also be interesting to apply the SBN balancer to adaptive mesh applications using a heterogeneous network of computers. Since low-cost processing power is readily available, it would be desirable to explore the effect of adding latency-tolerant techniques to the load balancing method. This approach would allow adaptive mesh applications to be executed with good results over low cost networks in contrast to homogeneous environments such as the IBM SP2. More precisely, this research includes how to adapt the processing to situations where some of processors in a network environment are not available. Fault tolerance would allow applications to make use of resources that are constantly changing during execution. Finally, the techniques presented could be applied to other applications such as multimedia image processing and data mining.

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

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

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

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

[17] 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.

[18] L. Oliker, R. Biswas, and R.C. 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.

[19] 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).

[20] 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.

[21] K. Schloegel, G. Karypis, and V. Kumar, "Multilevel diffusion schemes for repartitioning of adaptive meshes," Department of Computer Science, University of Minnesota, Minneapolis, MN (1997), Technical Report 97-013.

[22] 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.

[23] 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.

[24] C. Walshaw, M. Cross, and M.G. Everett, "Parallel dynamic graph-partitioning for unstructured meshes," School of Computing and Mathematical Sciences, University of Greenwich, London, UK (1997), Technical Report 97/1M/20.