Adaptive Load-Balancing Algorithms Using Symmetric Broadcast Networks

Sajal K. Das and Daniel J. Harvey
Department of Computer Sciences
University of North Texas
P.O. Box 13886
Denton, TX 76203-6886
E-mail:{das,harvey}@cs.unt.edu

Rupak Biswas
MRJ Technology Solutions
NASA Ames Research Center
Mail Stop T27A-1
Moffett Field, CA 94035-1000
E-mail: rbiswas@nas.nasa.gov

Abstract

In a distributed-computing environment, it is important to ensure that the processor workloads are adequately balanced. Among numerous load-balancing algorithms, a unique approach due to Das and Prasad defines a *symmetric broadcast network* (SBN) that provides a robust communication pattern among the processors in a topology-independent manner. In this paper, we propose and analyze three novel SBN-based load-balancing algorithms, and implement them on an SP2. A thorough experimental study with Poisson-distributed synthetic loads demonstrates that these algorithms are very effective in balancing system load while minimizing processor idle time. They also compare favorably with several other existing load-balancing techniques. Additional experiments performed with real data demonstrate that the SBN approach is effective in adaptive computational science and engineering applications where dynamic load balancing is extremely crucial.

Key words: Hypercube, job migration, load balancing, mesh adaptation, network topology, spanning binomial tree, system load, Poisson distribution.

1 Introduction

To maximize the performance of a multicomputer system, it is essential to evenly distribute the load among the processors. In other words, it is desirable to prevent, if possible, the condition where one node is overloaded with a backlog of jobs to be processed while another processor is lightly loaded or idle. The load-balancing problem is closely related to scheduling and resource allocation, and can be static or dynamic. A *static* allocation [23, 25] relates to decisions made at compile time, and compile-time programming tools are necessary to adequately estimate the required resources. On the other hand, *dynamic* algorithms [4, 10, 14] allocate/reallocate resources at run time based on a set of system parameters that are maintained. For example, these parameters determine when jobs can be migrated and account for the overhead involved in such a transfer [24]. Determining the parameters to be maintained and how to broadcast them are important design considerations. Distributed scheduling policies [12, 16] are used to resolve these issues.

In this paper, we consider general-purpose distributed-memory parallel computers in which processors (or nodes) are connected by a point-to-point network topology and the nodes communicate with one another using message passing. Responsibility for load balancing is decentralized, or spread among the nodes. Processor workload is determined by the length of the local job queue. The network is assumed to be homogeneous and any job can be processed by any node. However, jobs cannot be rerouted once execution begins.

Recently, Das et al. [7, 8, 9] have suggested a different approach to load balancing, by introducing a logical topology-independent communication pattern called a *symmetric broadcast network* (SBN). We refine this approach and propose three novel and efficient load-balancing algorithms, one of which is adapted for use on the hypercube architecture. Based on their operational characteristics, our SBN-based algorithms can be classified (e.g. [26]) as:

Adaptive: performance adapts to the average number of queued jobs;

Symmetrically Initiated: senders and receivers can initiate load balancing;

Stable: the network is not burdened with excessive load-balancing traffic;

Effective: System performance does not degrade while balancing loads.

The three algorithms proposed in this paper have been implemented on an IBM SP2 multiprocessor [1], using the Message-Passing Interface (MPI) [19]. Performance of the SBN algorithms are analyzed by extensively conducting two sets of experiments. The first set of experiments uses Poisson-distributed synthetic loads and compares the SBN algorithms to other existing techniques such as Random [10], Gradient [17, 18], Sender Initiated [11], Receiver Initiated [11], and Adaptive Contracting [11]. The second set of experiments applies the SBN approach to a dynamic mesh adaptation application using actual load data. Our experiments demonstrate that a superior quality of load balancing is achieved by the SBN approach with respect to such metrics as the total jobs transferred, total completion time, message traffic per node, and maximum variance in node idle time. Furthermore, results show that SBN techniques can be applied to adaptive mesh-based computational problems where dynamic load balancing is very important. A preliminary version of this paper is presented in [6].

This paper is organized as follows. Section 2 reviews several known approaches for load balancing that will be used for comparison purposes. Section 3 defines symmetric broadcast networks. Section 4 discusses general characteristics common to all of the proposed algorithms. Section 5 presents three SBN-based load-balancing schemes. This section also analyzes the performance characteristics of these algorithms, while Section 6 summarizes the experimental results comparing SBN algorithms to other load-balancing techniques. Section 7 discusses the effectiveness of the SBN approach in a dynamic mesh application. The final section concludes the paper.

2 Previous Work

Among various approaches suggested in the literature for comparing load-balancing algorithms, three categories of analysis predominate: (a) mathematical modeling, (b) solving well-known problems in a multiprocessor environment, and (c) simulation. For example, in [22], the probability of load-balancing success is computed analytically. In [13], several load-balancing methods are compared by implementing Fibonacci number generation, the N-Queens problem, and the 15-puzzle on a network. Several analyses have also employed the simulation approach [15].

In this paper we perform experiments on an IBM SP2 multiprocessor, using the simulation approach with synthetically-generated random loads according to Poisson distributions. However, mathematical analysis is also presented to provide a better understanding of algorithm performance. In addition, experiments that utilize actual load data from a dynamic mesh application are reported.

Many load-balancing algorithms that are compared are very susceptible to the choice of system thresholds [21]. A proper selection of threshold values has proven helpful in optimizing the SBN-based algorithms that we propose. The following load-balancing algorithms will be compared with ours using several performance metrics.

Random [10]:

Jobs are randomly distributed among processors or nodes.

As jobs are generated, and if the number of jobs queued at a given node is above a designated threshold, the jobs are randomly distributed between the originating and neighboring nodes. Once a job originating at a node is received by another node, it is processed. Therefore, additional job migration is not allowed. Single distribution messages can contain multiple jobs when more than one job is to be sent from one node to another.

Gradient [17, 18]:

Jobs proceed from overloaded to lightly-loaded nodes. This is accomplished by a systemwide gradient that is maintained. Each node has a load status flag. When compared with system thresholds, the value stored in this flag determines whether the node is overloaded, lightly loaded, or moderately loaded.

An array, PRESSURE, is also maintained at each node. This array has an entry that corresponds to each neighbor. Each of these entries contains the pressure (minimum number of communication "jumps" to the nearest lightly-loaded node) if a job is to be routed to the neighbor that corresponds to this entry. If a node i is lightly loaded, its pressure is zero. Otherwise it is calculated as:

$$\min \ \{\text{PRESSURE for each neighbor node of } i\} + 1.$$

Whenever the pressure of a node changes, it is broadcast to all of its neighbors. Because of network dynamics, this pressure is only an approximation of the true system load.

Under the gradient algorithm, a job can migrate many times before it is finally processed.

Receiver Initiated [11]:

Load balancing is triggered by a lightly-loaded node. If a given node has a load value below the system threshold, it broadcasts a job request message to its neighbors. The node's job queue length is "piggy backed" to the request message. Upon receipt of this message, each neighbor node compares its job queue length to that of the requesting node. If the local queue size is larger, the neighbor node replies with a single job.

To prevent instability in light system load conditions, a time-out of one second is introduced to wait for job replies. More specifically, the node will wait one second before initiating another request for jobs. It is possible for a job to be migrated multiple times using this algorithm before being processed.

Sender Initiated [11]:

Load balancing is initiated when nodes become overloaded. To prevent instability under heavy system loads, each node exchanges load information with its neighbors. Load values are exchanged when a local job queue size is halved or doubled in length. In this way, the exchange of load information occurs less frequently as the system load increases.

When jobs are generated, they are distributed to lightly-loaded neighbors. Once a job is received from a neighbor node, it is processed. Multiple job migrations are not allowed.

Adaptive Contracting [11]:

When jobs are generated, the originating node distributes bids to its neighbor nodes in parallel. The neighbor nodes respond to this bid with a message containing the number of jobs in their respective local queues.

The originating node then distributes jobs to those neighbors that have loads smaller than an amount determined by a system threshold. The number of jobs distributed is such that jobs are equally divided among the originating node and its lightly-loaded neighbors.

3 Preliminaries on Symmetric Broadcast Networks (SBNs)

A *symmetric broadcast network* (SBN) defines a communication pattern (logical or physical) among the P processors in a multicomputer system [7, 9]. 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 and additional interprocessor connections as follows:
  (a) Node i in stage 0 is connected to node $j = (i + P/2) \bmod P$ in stage 1; and
  (b) Node j in stage 1 is connected to the node in stage 2, that was the stage 0 successor of node i in SBN($d-1$).

An example of how an SBN(2) is formed from two SBN(1)s is shown in Fig. 1. The SBN approach defines unique communication patterns among the nodes in the network. For any source node at stage 0, there are $\log P$ stages of communication with each node appearing exactly once. The 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.

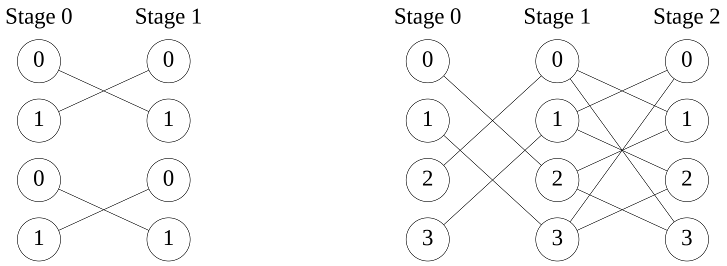


Figure 1: Construction of SBN(2) from a pair of SBN(1)s.

As an example, consider the two communication patterns for SBN(3) shown in Fig. 2. The paths in Fig. 2(a) are used to route messages originating from node 0, while those in Fig. 2(b)

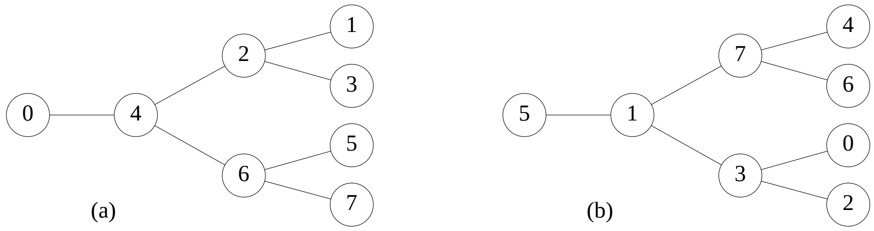


Figure 2: Examples of SBN communication patterns in SBN(8).

are for messages originating from node 5. Now if n_5^s denote a node at stage s in Fig. 2(b) and n_0^s is the corresponding node in Fig. 2(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 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 to n_0^s can be computed as follows:

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 $1 \leq s < d$.

Figure 2 illustrates two possible SBN communication patterns, but many others can easily be derived based on network topology and application requirements. For example, the SBN node 0 pattern can be defined based on the array implementation of a full binary tree. Predecessor and successors are then defined as follows:

Predecessor $= \left\lfloor \frac{n_0^s}{2} \right\rfloor$ if $s > 0$.

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

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

In [5], the SBN approach was adapted for use on the hypercube using a *modified binomial spanning tree*, which is actually two binomial trees connected back to back. Figure 3 shows such a communication pattern for a 16-node network which is used to route messages originating from node 0. The solid lines of the diagram represent the actual SBN pattern, whereas the dashed lines are used to gather load-balancing messages at a single destination node (node 15).

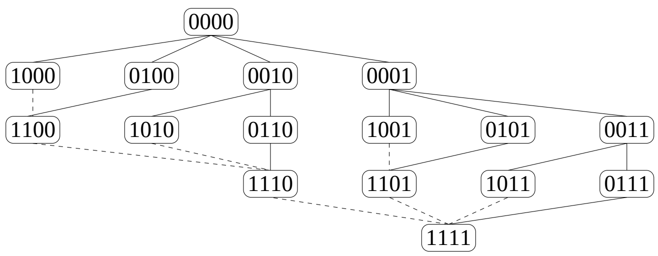


Figure 3: Binomial spanning tree used as a hypercube SBN.

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 $\max\{1, b\}$ where b is the number of consecutive leftmost 1-bits in the node's binary address.

4 General Characteristics of Proposed Load Balancing

4.1 System Thresholds

All SBN-based algorithms adapt their behavior to the system load. Under heavy (light) loads, the balancing activity is primarily initiated by processors that are lightly (heavily) loaded. This activity is controlled by two system thresholds, `MinTh` and `MaxTh`, which are respectively the minimum and maximum system load levels. The system load level `SysLL` is the average number of jobs queued per processor. If a processor has a queue length, `QLen`, below `MinTh`, a message is initiated to begin load balancing. If `QLen` is larger than `MaxTh`, extra jobs are distributed through the network. If this distribution overloads other processors, load balancing is triggered.

Algorithm behavior is affected by the values chosen for `MinTh` and `MaxTh`. For instance, `MinTh` must be large enough so that sufficient jobs can be received before a lightly-loaded processor becomes idle. However, the value should not be so large as to initiate unnecessary load balancing. If `MaxTh` is too small, it will cause an excessive number of job distributions. If it is too large, jobs will not be adequately distributed under light system loads. Moreover, once there is sufficient load on the network, very little load-balancing activity should be required.

4.2 Message Communication

Two types of messages are processed by the SBN approach. The first type is the balancing message which is sent through the network to indicate unbalanced system load. These messages are originated from an unbalanced node and then routed through the SBN. As these balancing messages pass through the network, the cumulative total of queued jobs is computed to obtain `SysLL`. The second message type for job distribution and is used for three purposes. First, they are used to route the `SysLL` through the network. Each node, upon receipt of such a message, updates its local values for `MinTh`, `MaxTh`, and `SysLL`. Second, job distribution messages are used to pass excess jobs from one node to another. This action can occur whenever a node has more jobs than its `MaxTh`. Third, jobs can be distributed when a node responds to another node's need for jobs. This need is embedded in both load-balance messages and in distribution messages.

If the communication from one node to its neighbors is completed in constant time, a single load-balancing operation requires $O(\log P)$ time since there are $d+1 = (\log P)+1$ communication stages in SBN(d). However, if multiple balancing operations are processed simultaneously, the worst case complexity is $O(\log^2 P)$ [7]. To reduce message traffic, a node does not initiate additional load-balancing activity until all previous balancing-related messages that have passed through the node have been completely processed.

4.3 Common Procedures

All of our load-balancing algorithms based on the SBN scheme consist of four key procedures. The first two, *GetDistribute* and *GetBalance*, are used to respectively process distribution and balance messages that are received. Similarly, the procedures, *Distribute* and *Balance*, respectively route distribution and balance messages to the SBN successor nodes. Details of these procedures depend on the particular load-balancing algorithms used. Figure 4 presents the pseudo code that is common to all of the SBN based load-balancing algorithms.

Procedure *Main Line Processing*

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

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

Figure 4: Common pseudo code for SBN based load-balancing algorithms.

5 SBN-Based Load-Balancing Algorithms

5.1 Standard SBN Algorithm

In the standard SBN algorithm, load-balancing messages are routed through SBN from the source to the processors at the last stage. Load-balance messages are then routed back towards the original source so the total number of jobs in the system can be computed. The originating node thus has an accurate value of `SysLL`. Distribution messages are then sent to all nodes along with `SysLL`. All nodes update their local `SysLL`, `MinTh`, and `MaxTh`. Excess jobs are routed as part of this distribution to balance the system load. In addition, if a processor has `QLen` less than `SysLL`, the need for jobs is indicated during the distribution process. Successor nodes respond by routing back an appropriate number of excess jobs. Figure 5 provides pseudo code of the standard SBN algorithm.

To illustrate the processing involved in a load-balancing operation, consider the SBN(3) in Fig. 6(a). The id and `QLen` for each node are shown. For example, node 6 has three jobs queued for

Procedure *GetBalance*
While there are balance messages to be processed
    Route any needed jobs to the predecessor node if possible
    If more balancing messages are still to be gathered, **Break**
    If this is the final SBN stage
        Route distribution and `SysLL` to originator node
    If this is the originator of the balancing operation
        Decrement the number of balance operations being processed
        Call *UpdateLoad*(`TotalJobsQueued`)
        Distribute excess jobs and `SysLL` through the SBN
    else Increment load-balancing operations being processed
        Route the balance message to the next SBN stage
End While
Return

Procedure *GetDistribute*
While there are distribution messages to be processed
    Enqueue any jobs received
    If the predecessor node has a need for jobs, route excess jobs back
    If load balancing is complete
        Decrement number of balancing operations being processed
        Call *UpdateLoad*(`TotalJobsQueued`)
    If this is message completes a distribution
        If the queue size > maximum threshold trigger load balancing
    else Call Distribute to route the excess jobs to the next SBN stage
end While
Return

Procedure *Balance*
If this is the final stage, **Return**
If this is a new balance operation
    If load balancing is in process, **Return**
    Increment the number of balance operations being processed
    Compute the number of distribution messages expected
Compute the number of jobs needed
Route the balance message to the next SBN stage
Return

Procedure *Distribute*
If this is the final SBN stage, **Return**
If this is a normal distribution and load balancing is being performed
    Inhibit the distribution and **Return**
Compute the number of excess messages and the number of needed jobs
Dequeue the jobs to be distributed
Distribute jobs and forward the `SysLL` data to successors
Return

Figure 5: Pseudo code for the standard SBN algorithm.

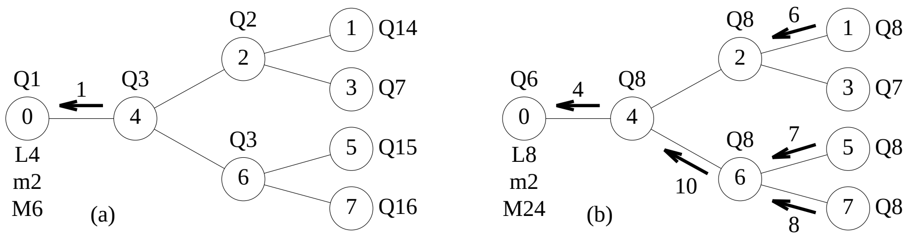


Figure 6: An example of load balancing using the standard SBN algorithm.

processing, indicated as $Q3$. The initial values of the `SysLL`, `MinTh`, and `MaxTh` at node 0 are 4, 2, and 6, respectively (indicated as $L4$, $m2$, and $M6$). After a load-balancing request is sent through the SBN and then routed back to node 0, these values are updated as 8, 2, and 24, respectively, using:

$$\texttt{SysLL} = \lceil \texttt{TotalJobsQueued} / P \rceil$$

$$\texttt{MinTh} = \min\{\texttt{ConstantParameter}, \texttt{SysLL} - 1\}$$

$$\texttt{MaxTh} = \texttt{SysLL} + 2^{\lfloor \texttt{SysLL/ConstantParameter} \rfloor}$$

Note that when the balancing is initiated, node 4 distributes half of its `QLen` jobs, i.e. $\lfloor 3/2 \rfloor$, back to node 0 which had a need for jobs. This distribution is shown by a label on the arrow in Fig. 6(a).

Distribution messages are then used to route excess jobs to the successor nodes or to indicate a need for jobs if the local `QLen` is less than `SysLL`. Jobs are routed back to the predecessor nodes when appropriate. Figure 6(b) shows the result of this distribution. The arrows indicate the number of jobs routed between nodes.

To load balance P processors, $P-1$ balance messages are sent through the SBN. Then $P-1$ distribution messages are routed back to the originating node with the `SysLL` value. Finally, another $P-1$ distribution messages are sent to complete the operation. Thus, a total of $3P-3$ messages have to be processed, requiring a total time of $O(\log P)$ for this operation.

5.2 Hypercube Variant

The SBN approach can be adapted for implementation on a hypercube topology, using the modified binomial spanning tree sketched in Section 3. A complete description of this hypercube variant is given in [5]. It operates in a manner similar to the standard SBN algorithm with the following differences:

- The value of `SysLL` is computed when all balance messages arrive at the destination node in the network. This is possible because there is a unique destination node for every originating node. Distribution messages are then routed back to complete the load balancing. Since there are $P - 1 + \frac{P}{2} - 1$ interconnections in the modified binomial spanning tree (cf. Fig. 3), a load-balancing operation requires $3P-4$ messages to be processed.

- Nodes in the SBN need to gather all balancing messages from their predecessors before routing the updated `SysLL` to the successors.

- The network topology is such that the number of predecessor and successor nodes vary at the different stages of communication.

5.3 Heuristic SBN Algorithm

Both of the previous algorithms are expensive since a large number of messages has to be processed to accurately maintain the `SysLL`. The heuristic version attempts to reduce the amount of processing by terminating load-balancing operations as soon as enough jobs are found that can be distributed. In general, this strategy reduces the number of messages; although $O(P)$ messages are needed in the worst case.

In the heuristic algorithm, a processor estimates `SysLL` by averaging `QLen` for the processors through which the balance message has passed. An appropriate number of jobs is then returned to the predecessor nodes as follows:

$$\texttt{ExJobs} = \begin{cases} 0 \quad \text{if} & \texttt{QLen} < 3 \\ \lfloor \texttt{QLen}/2 \rfloor & \text{otherwise.} \end{cases}$$

If `ExJobs` $= 0$ or if `SysLL` > 2 when `ExJobs` $= 1$, the balance message is forwarded to the next stage. Otherwise, the load balancing is terminated. The justification for this strategy is discussed in Section 6.

Job distribution is also processed differently in the heuristic SBN algorithm. For example, consider the network SBN(3) that has a processor with `MaxTh`$=15$ and `QLen` $= 24$. The number of jobs to be distributed is computed by dividing `QLen` by the total number of stages. Thus, six jobs are distributed in this case. `SysLL` is then set to $24 - 6 = 18$. The processor that receives these jobs divides the number of jobs received by the remaining number of stages and adds the result to the `SysLL` stored at that node. The pseudo code in Fig. 7 gives the operational details of the heuristic SBN algorithm.

5.4 Remarks

A significant advantage of the heuristic variant is that the load-balancing messages do not have to be gathered until `SysLL` can be estimated. This reduces the interdependencies associated with the communication. If a particular processor fails, load balancing can still be accomplished utilizing the remaining processors.

An additional improvement has been obtained for all three load-balancing algorithms by using using multiple SBN communication patterns. Each time a message is initiated, one of the SBN patterns is randomly chosen. Each message includes the source node, the pattern used, and the stage to which the message is being routed. Since all nodes have the SBN template associated with messages originating from node 0, the required SBN communication pattern can be determined. Multiple randomly-selected SBN patterns distribute messages more evenly, enhance network reliability, and allow various applications to be written using different communication patterns.

5.5 Mathematical Analysis

In a network of P processors, the distribution of jobs among processors can be modeled using a Poisson distribution. Specifically, the probability of a given node having j jobs is $\frac{\lambda^j}{e^\lambda j!}$, where λ is the mean arrival rate. If the system load level `SysLL` is k, then, by definition, the average number of jobs assigned to a processor is k. Hence, the probability that a node has j jobs is $\frac{k^j}{e^k j!}$. Using this simple model, useful probabilities can be easily calculated. For example, the probability, g_3, that a processor in the network has more than three jobs, is $g_3 = 1 - \frac{1}{e^k}(1 + k + \frac{k^2}{2} + \frac{k^3}{6})$. The probability that all P processors have more than three jobs is g_3^P. Now, $g_3^P > 0.9$ if $k > 5$ and is

```
Procedure GetBalance
While there are balance messages to be processed
        Calculate the estimated TotalJobsQueued
        Call UpdateLoad(TotalJobsQueued)
        Distribute excess jobs to the predecessor node
        If no jobs distributed or one job distributed when SysLL > 2
                Route the balance message to the next SBN stage
End While
Return

Procedure GetDistribute
While there are distribution messages to be processed
        If this distribution is in response to load balancing
                NewLL = SysLL + ⌈ JobsReceived/(Stage +1)⌉
        else    NewLL = QLen + ⌈JobsReceived/(2^(Dim−Stage) − 1)⌉
        Call UpdateLoad(P× NewLL)
        Enqueue received messages
        Continue the distribution to the next SBN stage
End While
Return

Procedure Balance
If this is the final stage, return
Route the Balance message to the next SBN stage
Return

Procedure Distribute
If this is the final SBN stage, return
If this is a response to a load-balancing operation
        If (QLen < 3)
                ExJobs = 0
        else    ExJobs = ⌊Qlen/2⌋
else    ExJobs = QLen − MaxTh
If the last job is to be distributed, ExJobs = 0
Dequeue the jobs to be distributed
Distribute the ExJobs among the adjacent SBN nodes
Return(NumberOfJobsDistributed)
```

Figure 7: Pseudo code for the heuristic SBN algorithm.

almost unity if $k > 15$. This implies that the need for load-balancing activity rapidly decreases as `SysLL` increases. Therefore, increasing `MaxTh` exponentially as `SysLL` increases makes good sense.

In order to analyze the heuristic algorithm, we need to analyze other network characteristics such as (a) the expected number of stages through which load-balancing operations must travel before at least two jobs are returned and (b) the expected number of jobs that will be returned as a result of such an operation.

If J_i is the probability that i jumps are required, the expected number of jumps is $E_j = \sum_{i=1}^{\log P} i * J_i$. Assuming that a load-balancing operation stops when two or more jobs are returned, the first node with at least four jobs will terminate the load-balancing process when using the heuristic algorithm. Since g_3 is the likelihood of a node having more than three jobs to process, we can evaluate the above sum. Unfortunately, the expressions needed to calculate J_i become quite complicated as i increases. However, because $i \leq \log P$, we have evaluated this equation for networks up to 64 nodes. Our analysis shows that the number of expected jumps becomes very close to unity when $k > 7$. This conclusion confirms that the heuristic algorithm should greatly reduce the number of messages that need to be processed for networks with heavy system loads.

The handling of light system loads is an entirely different matter. For example, in an SBN(5), an average 4.931 stages need to be processed if $k = 1$. With further analysis, we found that balancing operations on an average need to pass through 28.265 of the 32 nodes. Therefore, effective load balancing when processing light system loads is expensive. This result motivated us to impose the constraint that the algorithm should stop load balancing as soon as a single job is returned when the system is lightly loaded.

Finally, in order to establish how many jobs, on average, will be returned in response to a load-balancing message requires three parameters: (a) the probability that a single job will be returned (`QLen` = 3), (b) the probability that at least two jobs will be returned (`QLen` > 3), and (c) the number of nodes through which a balancing operation must pass. We have found that in a network of 32 nodes, if the `SysLL` = 2, an average of 7.493 jobs will be returned. Again, requiring that the algorithm stops load-balancing operations as soon as a single job is returned when processing light loads, alleviates this situation. The number of expected jobs that are returned is effectively reduced to a reasonable value.

6 Testing Procedures and Experimental Results

6.1 Simulation Environment

The three SBN-based load-balancing algorithms have been implemented using MPI and tested with synthetically-generated workloads on the SP2 located at NASA Ames Research Center. The simulation program spawns the appropriate number of child processes and creates the desired network. The list of all process ids and an initial distribution of jobs is routed through the network.

In addition to the initial load, each node dynamically generates additional job loads to be processed. Namely, 10 job creation cycles are processed. The number of jobs generated at each node during each cycle follows a Poisson distribution. By randomly picking different values of λ, varying numbers of jobs are created. Therefore, both heavy and light system load conditions are dynamically simulated. Jobs are processed by "spinning" for the designated time period. The simulation terminates when all jobs have been processed. Three test runs are reported here:

Heavy System Load (cf. Fig. 8): Initially, 10 jobs per node are randomly distributed throughout the network. The jobs generated during execution are more than the network can process.

Job duration averages one second.

Transition from Heavy to Light System Load (cf. Fig. 9): Fifty jobs multiplied by the number of processors are distributed to a small subset of nodes as an initial load. A light load of jobs is generated as the load-balancing algorithm is processed. Job duration averages two seconds. Note that the initial load imbalance needs to be corrected.

Light System Load (cf. Fig. 10): A small number of jobs are initially distributed to a small subset of nodes. A light load of jobs are created as the algorithms execute.

The performance of the SBN based algorithms are compared with several popular algorithms (e.g. Random, Gradient, Sender Initiated, Receiver Initiated, Adaptive Contracting). The same simulation tests are also run without load balancing.

6.2 Performance Metrics

The data and line charts included in Figs. 8-10 measure the comparative performance of the various load-balancing algorithms on an SP2. The X-axis of the line charts show the number of processors used. The Y-axis tracks the following variables:

(a) **Message Traffic Comparison by Node**: Measures the maximum total number of load-balancing messages that were sent by any one of the nodes.

(b) **Total Jobs Transferred**: Measures the total number of job transfers that occurred from one node to another.

(c) **Maximum Variance by Node in Idle Time**: Measures the difference in processing time between the most busy node and the least busy node.

(d) **Total Time to Complete**: Measures the total amount of elapsed time in seconds before all jobs are fully processed.

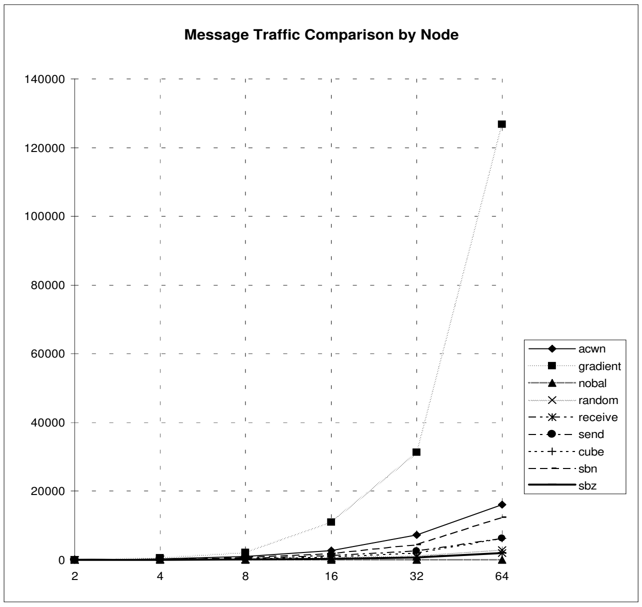
6.3 Summary of Results

As expected, the program with no load balancing (*nobal*) performs by far the worst. The *random* algorithm, although providing significant improvement in minimizing idle time, nevertheless is less effective than the remaining algorithms.

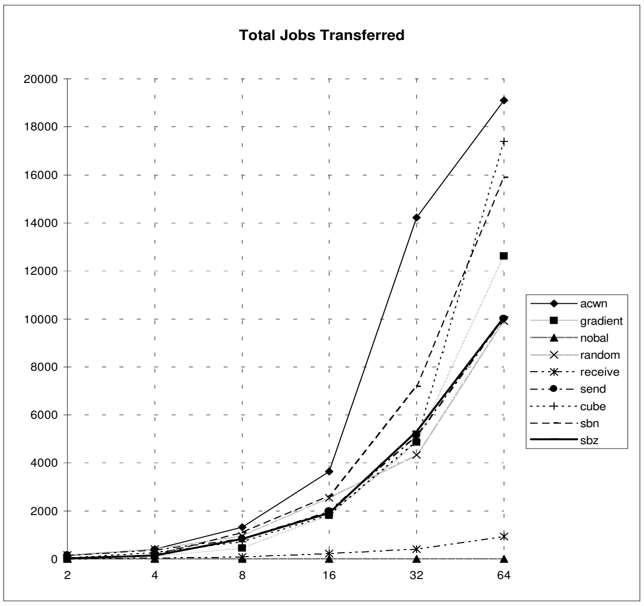
The Sender Initiated (*send*) algorithm more evenly balances the load than *random*; however, the Receiver Initiated (*receive*) algorithm does better only when the system load is light. For light to moderate loads, *receive* generates more network traffic because all nodes poll neighbors to find jobs they can process. To overcome this deficiency, a time delay of one second has been introduced after a polling operation at the cost of increasing the idle time. At heavy system loads *send* can cause job thrashing. This has been overcome by reducing the number of job transfers that are done at high load levels. However, it can cause one or more nodes to remain lightly loaded.

The Gradient (*gradient*) algorithm balances the load quite well without any of the above deficiencies. Unfortunately, lightly-loaded nodes can sometimes receive too many messages from the overloaded nodes. Also, message communication required to update neighbor node information is

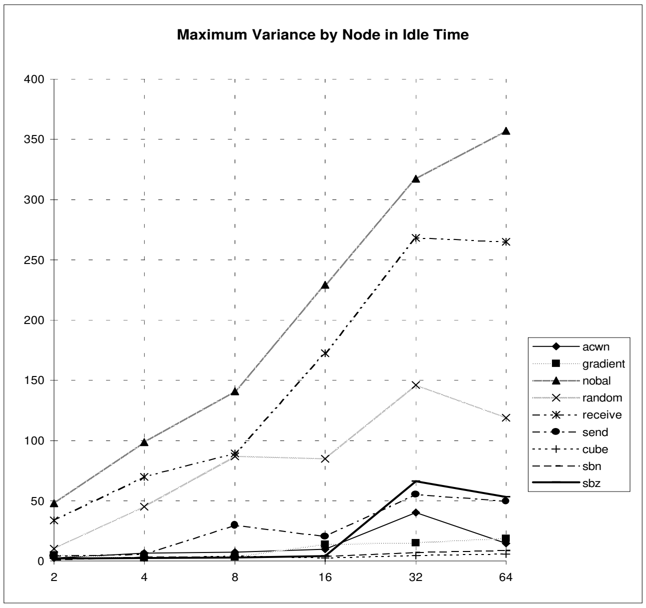
processors	acwn	gradient	nobal	random	receive	send	cube	sbn	sbz
2	96	50	0	15	14	34	9	12	2
4	294	586	0	74	80	136	77	68	18
8	1008	2093	0	198	180	439	285	678	134
16	2709	10915	0	560	484	1279	906	1882	320
32	7247	31232	0	1190	864	2617	1980	4420	680
64	16043	126807	0	2810	1999	6290	6233	12434	1942



processors	acwn	gradient	nobal	random	receive	send	cube	sbn	sbz
2	141	26	0	158	7	148	34	31	28
4	406	127	0	414	40	373	264	160	146
8	1324	454	0	949	85	837	719	1092	830
16	3648	1820	0	2539	228	1979	1824	2619	1905
32	14223	4856	0	4335	412	5139	4923	7182	5315
64	19113	12616	0	9907	930	10005	17405	15900	10080



processors	acwn	gradient	nobal	random	receive	send	cube	sbn	sbz
2	2.5	4.9	47.7	10.1	33.5	4.1	2.3	0.9	2.0
4	6.4	2.3	98.4	45.0	69.7	5.3	2.4	3.1	2.6
8	7.5	3.2	140.9	87.0	89.2	29.5	4.0	3.2	3.0
16	9.9	13.7	229.4	84.8	172.6	20.2	2.5	3.6	4.0
32	40.3	15.0	317.2	146.2	268.3	55.3	4.3	6.7	66.3
64	14.6	18.6	356.9	118.8	264.9	49.6	5.5	8.4	53.3



processors	acwn	gradient	nobal	random	receive	send	cube	sbn	sbz
2	176.1	157.4	178.8	163.8	171.7	152.9	141.2	155.4	158.5
4	152.9	152.6	191.6	185.4	170.5	154.8	151.2	153.2	140.9
8	176.6	162.1	194.1	211.4	178.7	178.0	172.7	161.9	147.2
16	185.4	168.2	300.0	236.8	277.7	171.3	170.9	164.2	161.2
32	221.8	201.2	387.4	264.5	365.9	226.7	186.9	200.5	224.3
64	189.9	192.5	389.8	242.4	377.7	212.4	187.5	189.6	192.3

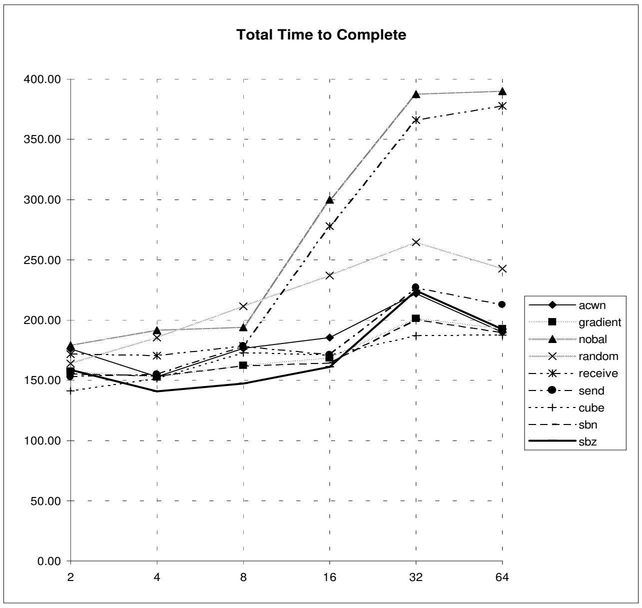
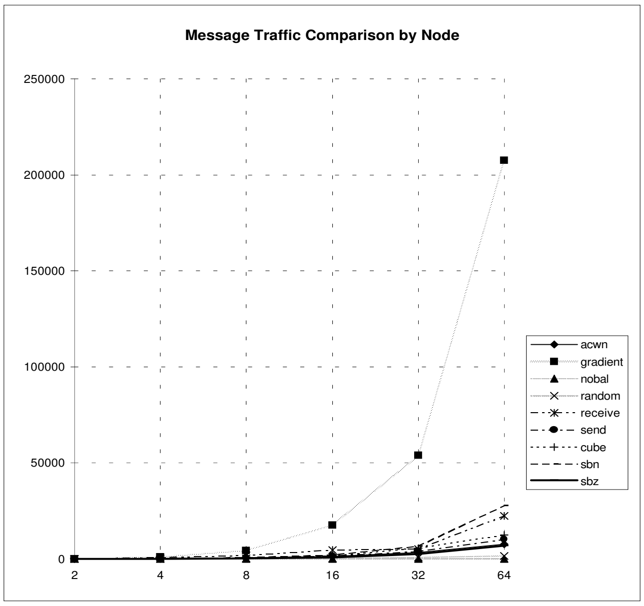
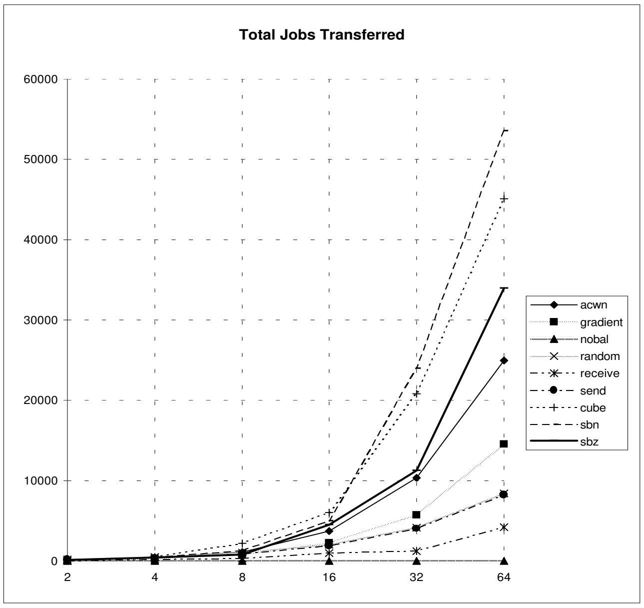


Figure 8: Heavy system load.

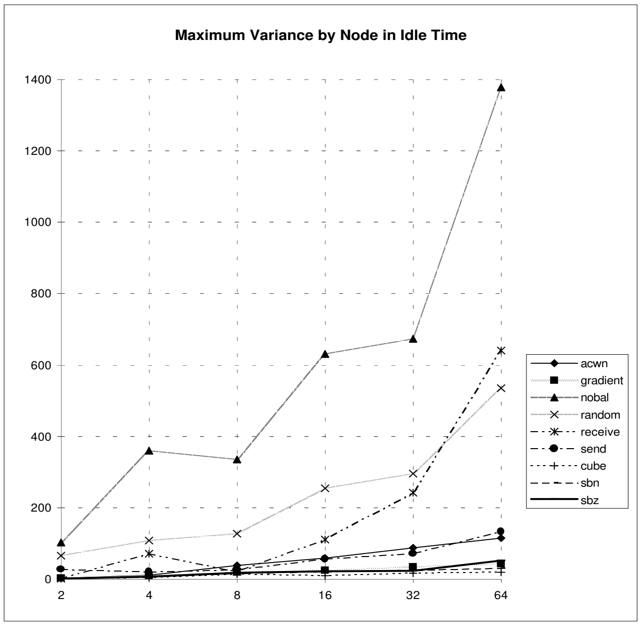
processors	acwn	gradient	nobal	random	receive	send	cube	sbn	sbz
2	58	40	0	13	51	25	15	12	7
4	198	979	0	67	786	191	110	166	90
8	515	4348	0	151	1699	540	512	692	228
16	1268	17498	0	347	4630	1568	1414	2127	802
32	3239	53961	0	804	5036	3767	5540	6704	2461
64	7713	207489	0	1649	22424	10120	12408	27772	6951



processors	acwn	gradient	nobal	random	receive	send	cube	sbn	sbz
2	133	23	0	181	24	207	141	118	109
4	398	284	0	496	174	345	503	404	406
8	1083	810	0	1011	334	878	2201	1264	786
16	3685	2265	0	2017	986	1880	6034	4997	4519
32	10355	5693	0	4121	1192	4002	20812	23958	11251
64	24947	14520	0	8417	4184	8217	45107	53580	33947



processors	acwn	gradient	nobal	random	receive	send	cube	sbn	sbz
2	3.1	3.3	102.3	65.9	1.1	27.3	1.9	1.7	1.9
4	10.9	9.4	360.2	108.5	70.6	20.3	4.6	6.0	6.7
8	38.0	19.2	335.2	127.8	20.6	25.6	14.3	14.9	17.9
16	59.2	25.2	630.5	254.9	111.5	56.3	9.6	20.3	21.5
32	87.7	34.4	673.4	295.6	241.9	71.0	16.4	24.6	23.3
64	115.4	42.1	1,378.5	534.6	640.2	132.5	19.0	30.1	52.2



processors	acwn	gradient	nobal	random	receive	send	cube	sbn	sbz
2	437.2	466.8	516.3	387.3	465.7	478.8	484.3	464.3	459.9
4	340.0	330.1	569.7	429.7	346.9	332.8	343.4	328.0	309.5
8	319.7	309.4	480.4	415.1	314.3	315.5	327.6	310.4	301.8
16	317.7	317.4	630.6	442.6	347.5	338.3	348.1	315.6	328.6
32	363.1	338.8	630.6	477.4	365.9	350.8	327.0	330.3	339.8
64	372.6	331.7	1,378.2	654.3	786.4	396.4	332.7	326.4	330.6

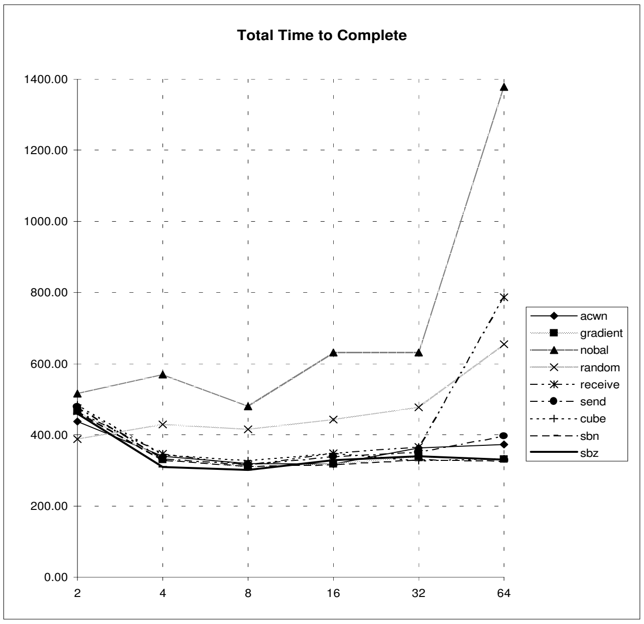
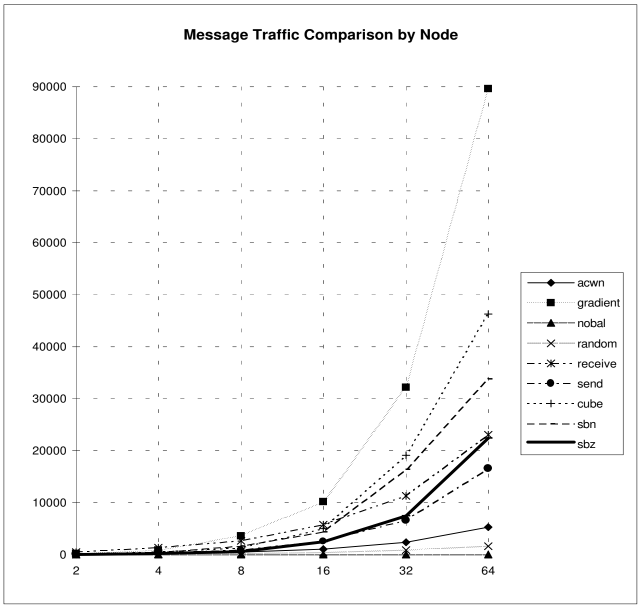
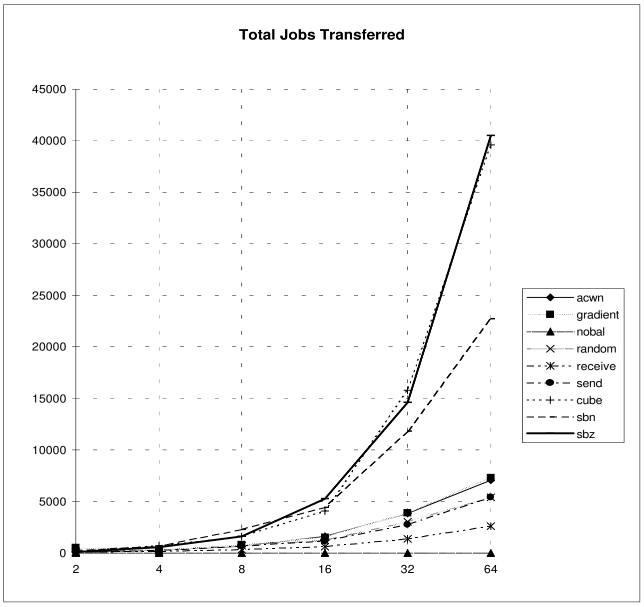


Figure 9: Transition from heavy to light system load.

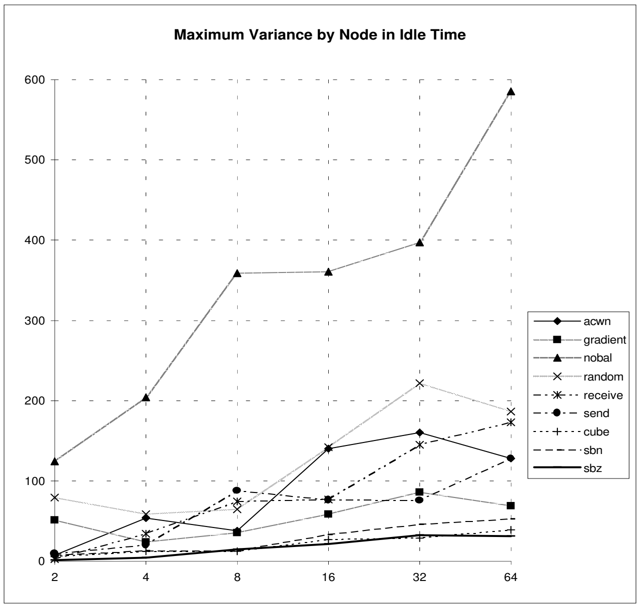
processors	acwn	gradient	nobal	random	receive	send	cube	sbn	sbz
2	28	172.02	0	16	498	75	80	94	23
4	191	862	0	54	1306	274	469	353	179
8	464	3618	0	144	2609	877	1256	1553	598
16	1073	10162	0	333	5783	2503	4989	4365	2403
32	2368	32184	0	810	11318	6568	19058	16356	7462
64	5292	89605	0	1603	23072	16579	46250	33719	22377



processors	acwn	gradient	nobal	random	receive	send	cube	sbn	sbz
2	49	520	0	139	91	118	131	258	110
4	274	285	0	301	140	247	690	645	561
8	663	784	0	689	344	653	1590	2293	1614
16	1595	1508	0	1227	627	1174	4081	4426	5277
32	3852	3907	0	3029	1393	2762	15809	11819	14592
64	7070	7292	0	5420	2626	5410	39601	22691	40512



processors	acwn	gradient	nobal	random	receive	send	cube	sbn	sbz
2	7.4	50.9	124.3	79.0	2.7	9.1	5.2	7.1	1.5
4	53.5	23.9	204.2	58.6	34.3	20.4	12.1	12.9	4.5
8	37.8	35.5	358.9	64.7	74.5	87.5	12.3	12.7	14.5
16	140.3	58.2	360.4	141.9	76.9	76.3	26.7	32.9	21.2
32	160.2	86.0	397.1	221.5	145.2	75.8	28.6	45.5	32.0
64	128.0	68.7	585.4	186.4	172.7	127.8	39.3	52.1	31.1



processors	acwn	gradient	nobal	random	receive	send	cube	sbn	sbz
2	132.8	133.0	322.0	317.2	258.5	264.4	219.1	256.3	266.9
4	268.2	211.7	321.5	245.4	216.2	209.5	252.4	204.7	230.4
8	243.6	241.9	360.5	267.6	252.4	267.3	251.9	226.2	231.0
16	295.1	231.5	360.5	254.8	234.4	239.0	184.0	206.4	229.5
32	297.1	254.0	397.2	321.8	275.3	250.7	253.7	242.8	234.3
64	274.0	247.7	585.3	306.1	311.1	294.3	275.5	232.5	238.6

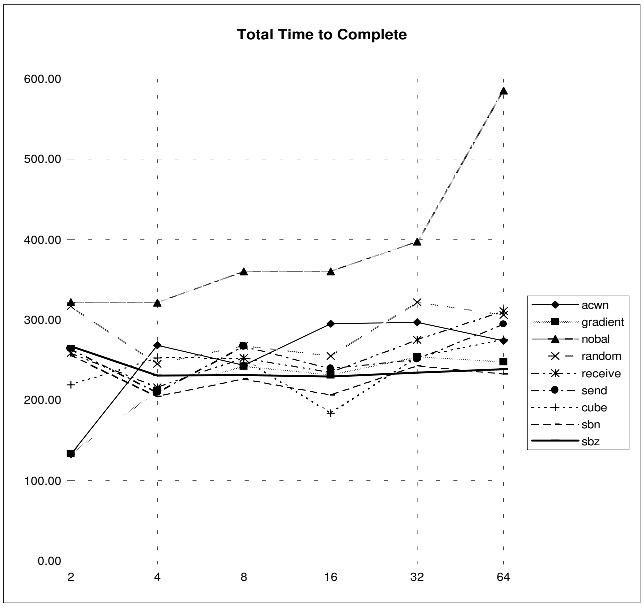


Figure 10: Light system load.

significant and often results in excessive network traffic. The Adaptive Contracting (*acwn*) algorithm performs the best in periods of heavy system loads. However, as was true for the *gradient* algorithm, an increased system traffic and the number of jobs migrated is observed.

Both the standard SBN (*sbn*) algorithm and its hypercube variant (*cube*) were able to balance the system load more evenly than other algorithms. Their performance characteristics are very similar. They require less message traffic than the *gradient* algorithm but cause a higher number of job migrations, especially in periods of light system loads.

The heuristic SBN algorithm (*sbz*) performs well in minimizing idle time in light system loads. Although its performance during periods of heavy loads is relatively good, it does not balance the generated system load as well as the *cube* or *sbn*. This is because its estimate of `SysLL` is not necessarily accurate. Note that for light loads, *sbz* requires many more job transfers than the other algorithms. However, it consistently requires fewer messages than *gradient*, *sbn*, or *cube*.

7 Applications to Dynamic Mesh Adaptation Problems

Numerical solution schemes for problems in computational science and engineering are usually performed on a mesh of vertices and edges. These applications can be modeled in our dynamic load-balancing framework as a grid of jobs where adjacent jobs need to communicate to complete their tasks. The traditional approach to such problems is to find a near-optimal minimal cut that partitions the grid among the processors in a network while balancing the individual processor workloads. For applications that require dynamic mesh adaptation, the load balancing procedure has to be invoked whenever the system load becomes unbalanced. Load balancing involves both partitioning the computational mesh and mapping the resulting partitions to the processors. Mapping requires data to be redistributed, that is, moved from one processor to another as determined by the partitioner.

Several load-balancing algorithms have been designed specifically for adaptive-mesh applications. In this work, we modify the standard SBN algorithm to make it applicable to such problems. Results obtained from our experiments show that it is an effective load-balancing technique for applications that undergo dynamic remeshing. Some of the major modifications that were made are the following:

(a) The unit of time required to execute a given job is determined by actual load data. The cost in time to distribute a job's data set from one processor to another is assumed to be equal to the processing time. This cost is incurred when a job begins to be executed on a processor different from the one to which the job was originally assigned. Note that the time to complete a job at a remote processor is at least double the cost of running that job on its original processor. Lastly, units of communication time, $Ctime$, are calculated according to the following formula:

$Ctime = \sum_{i=1}^{a} PR_i$,

where PR_i is the percentage of time that the data for adjacent job, i, resides at a remote node, and a is the number of adjacent jobs.

Note that job completion requires one extra unit of time for inter-job communication, if an adjacent job's data set resides at a remote processor during the entire run.

(b) The SBN algorithm utilizes multiple heaps (i.e. priority queues) to decide which jobs to process and which jobs to migrate. All jobs queued for processing are stored in a single 'local processing heap' and in one of a group of 'local migration heaps'. One local migration heap exists for each processor in the network. Table 1 illustrates this concept by listing a group of jobs queued for processing at node 0. Jobs 1 and 2 are on migration heap 0. Jobs 3-5 are stored on migration heap 2, 3, and 1, respectively. Note that the migration heap to which a job is placed corresponds to the processor at which the corresponding data set resides. By using this data structure, the SBN algorithm can quickly favor migration of jobs that are to be moved to the processor where the job's data is stored. For example, if according to Table 1, a job is to be distributed to node 3, job 4 will be favored since its data is stored at node 3. The balancing algorithm can quickly remove that job from the corresponding migration heap. Also the algorithm can efficiently choose jobs to migrate that would incur minimum increase in the system distribution and communication costs when the job is run. The heap order reflects this calculation. Lastly, the processing heap is used to choose jobs to be processed that would be the most expensive to migrate.

Table 1: Example of jobs queued for processing at node 0.

Jobs	Data Set
1	0
2	0
3	2
4	3
5	1

(c) A weighted system load level `WSysLL` is used rather than determining the system load based solely on the queue length. `WSysLL` accounts for the total processing time, communication time, and distribution time required to complete the running of the jobs in the system. Assume that $JDist_j$, $Radj_j$, and $Ladj_j$ respectively represent the distribution cost, the number of adjacent jobs that are in remote processors, and the number of adjacent jobs that are in the same processor for job, j. The value of `WSysLL` is then computed as follows:

$$\mathtt{WSysLL} = P \times \sum_{j=1}^{\mathtt{QLen}} (P \times JDist_j - Radj_j + P \times Ladj_j)$$

(d) Migration messages have been added to simulate the movement of data sets from one processor to another. Using the SBN, a migration message is broadcast when a job is about to be run and its corresponding data set is not resident. A locate message then is broadcast to indicate the new location of the data set. Therefore, all processors can maintain the processor location of all data sets.

(e) Minimum and maximum thresholds, `MaxTh` and `MinTh`, have been altered to reflect the time that will pass when all locally queued jobs are processed. Also, a limit has been placed on the number of jobs that can be migrated at once. This limit has the purpose of preventing excessive job distributions.

7.1 Experimental Results

Two experiments have been run using a dynamic-mesh adaptation load data. The computational mesh is one that was used to simulate an acoustics experiment where a 1/7th-scale model of a

UH-1H helicopter rotor blade was tested over a range of subsonic and transonic hover-tip Mach numbers. The initial mesh consisted of 13,967 vertices, 60,968 tetrahedral elements, and 78,343 edges. Numerical results and a detailed report of the simulation are given in [3, 27]. A total of three adaptations were performed on this initial mesh using a solution-based dynamic mesh adaptation procedure [2]. The final mesh contained 137,474 vertices, 765,855 tetrahedra, and 913,412 edges. However, load balancing was always performed on the initial mesh by modifying the weights of the vertices of the corresponding dual graph to model the dynamic mesh adaptation [20].

The results of these experiments are shown in Tables 2-3 and in Figs. 11-12. In the experiment charted in Table 2 and Fig. 11, we favor migration of jobs with long execution times. In the second experiment with results charted in Table 3 and Fig. 12, we favor migrating jobs with short execution times.

Table 2: Dynamic-mesh experiment favoring migration of long jobs.

Nodes	Jobs Migrated	Total Messages	Processing Percentage	Idle Percentage	Distribution Percentage	Communication Percentage
2	3,567	9,068	92.85	0.03	5.18	1.93
4	6,745	76,555	89.27	0.15	6.49	4.09
8	12,767	426,169	84.22	0.41	8.22	7.15
16	128,780	2,759,716	79.56	0.70	8.45	11.28
32	1,261,500	15,915,048	76.05	1.33	11.11	11.51
64	10,854,794	96,184,273	72.25	2.89	13.33	11.53

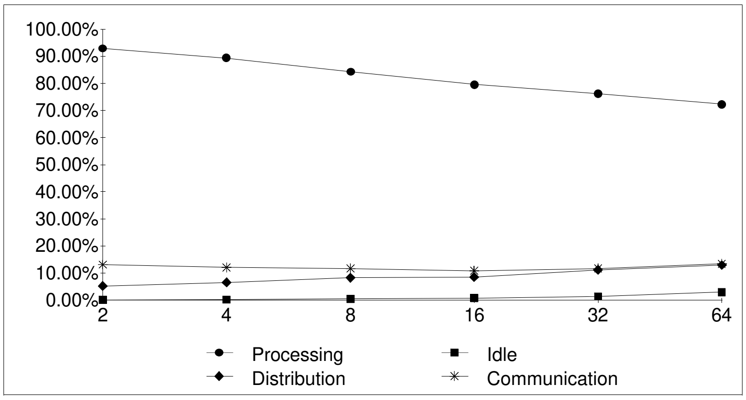


Figure 11: Dynamic-mesh experiment favoring migration of long jobs.

The results show that both experiments balance the load in such a way that minimal idle time is achieved. Favoring migration of large jobs increases the percentage of time spent processing jobs and lowers the number of jobs migrated. However, this improvement requires significantly more message traffic. If these results are extrapolated, minimal changes in idle time percentage occurs as the number of processor nodes are increased. Both experiments demonstrate that the SBN approach is general enough to be applicable to this class of problems.

Table 3: Dynamic-mesh experiment favoring migration of short jobs.

Nodes	Jobs Migrated	Total Messages	Processing Percentage	Idle Percentage	Distribution Percentage	Communication Percentage
2	29,298	58,531	82.89	0.02	13.04	4.05
4	53,636	80,991	80.15	0.03	12.03	7.79
8	84,225	514,549	76.15	0.12	11.63	12.10
16	182,036	2,074,743	72.80	0.51	10.77	15.92
32	913,601	7,139,339	70.87	0.92	11.59	16.62
64	3,603,769	19,858,167	68.32	1.34	12.88	17.46

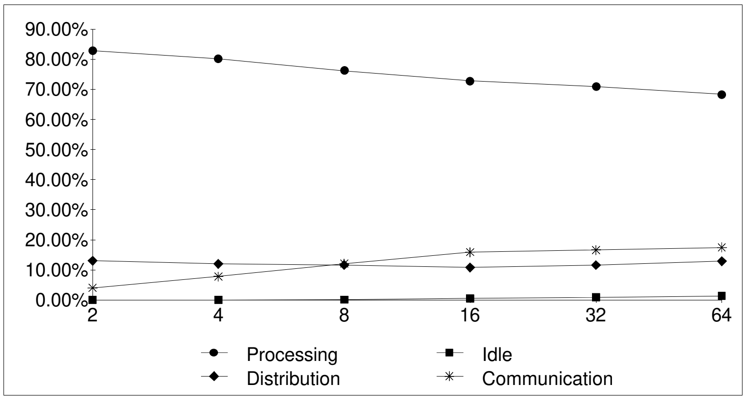


Figure 12: Dynamic-mesh experiment favoring migration of short jobs.

8 Conclusions

Empirical results have shown that our approach to load balancing using the concept of a symmetric broadcast network (SBN) is effective and superior to several other schemes. All three algorithms that we propose successfully balance the system load and minimize processor idle time. In addition, the heuristic variant reduces the overhead associated with load-balancing message traffic. We have also demonstrated that the SBN approach, when applied to a dynamic mesh application, is effective in minimizing the required communication and distribution costs. In our dynamic mesh experiments, while it may appear that the reduction in processor idle time is at a cost of a larger number of message transfers, we are currently developing heuristics with a goal to reduce such message transfers significantly. We expect to report our findings in the final version.

The research presented in this paper could be extended in different directions. Further adaptations of our SBN-based load balancing approach to a wide variety of topological interconnections (and hence multicomputer configurations) would make our scheme even more versatile and architecture-independent. This simply means how effectively SBNs can be mapped onto existing topologies like meshes, fat trees, etc. In sections 3 and 5.2, we have demonstrated the mapping of SBNs on a hypercube topology. Therefore, with the help of binary reflected Gray codes, it is straightforward to embed SBNs into meshes. Another important area for research is to analyze the effect of altering the definition of "system load". In the standard SBN algorithm, we have assumed that the local queue size determines system load, whereas in the dynamic-mesh adaptation, a weighted queue length was used. However, other parameters such as processor resource allocation and execution dependencies could greatly alter how load balancing should be accomplished.

References

[1] T. Agerwala, J. Martin, and J. Mirza, "Research Report SP2 System Architecture," *IBM Corporation Publication 95A001198*, 1995.

[2] R. Biswas and R.C. Strawn, "A New Procedure for Dynamic Adaption of Three-Dimensional Unstructured Grids," *Applied Numerical Mathematics*, Vol. 13, No. 6, pp. 437–452, Feb. 1994.

[3] R. Biswas and R.C. Strawn, "Mesh Quality Control for Multiply-Refined Tetrahedral Grids," *Applied Numerical Mathematics*, Vol. 20, No. 4, pp. 337–348, Apr. 1996.

[4] G. Cybenko, "Dynamic Load Balancing for Distributed-Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 7, No. 2, pp. 279–301, Oct. 1989.

[5] S.K. Das and D.J. Harvey, "Performance Analysis of an Adaptive Symmetric Broadcast Load Balancing Algorithm on the Hypercube," *Technical Report*, Department of Computer Science, University of North Texas, 1995.

[6] S.K. Das, D.J. Harvey, and R. Biswas, "Adaptive Load-Balancing Algorithms Using Symmetric Broadcast Networks: Performance Study on an SP2," *Proceedings of the International Conference on Parallel Processing*, Bloomingdale, Illinois, Aug. 1997, to appear.

[7] S.K. Das and S.K. Prasad, "Implementing Task Ready Queues in a Multiprocessing Environment," *Proceedings of the International Conference on Parallel Computing*, Pune, India, pp. 132–140, Dec. 1990.

[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," *Technical Report*, CRPDC-92-1, Department of Computer Science, University of North Texas, 1992.

[9] S.K. Das, C-Q. Yang, and N.K. Leung, "Implementation of Load Balancing in Multiprocessor Systems Using a Symmetric Broadcast Network," *Proceedings of the International Conference of Parallel and Distributed Systems*, Hsinchu, Taiwan, pp. 589–596, 1992.

[10] D.L. Eager, G. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 5, pp. 662–675, 1986.

[11] D.L. Eager et al., "A Comparison of Receiver Initiated and Sender Initiated Adaptive Load Sharing," *Performance Evaluation*, Vol. 6, pp. 63–68, 1986.

[12] M.R. Eskicioglu, "Design Issues of Process Migration Facilities in Distributed Systems," *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press, Los Alamitos, CA, pp. 414–424, 1995.

[13] M.D. Feng and C.K. Yuen, "Dynamic Load Balancing on a Distributed System," *Proceedings of the Symposium on Parallel and Distributed Processing*, Dallas, TX, pp. 318–325, Oct. 1994.

[14] G. Fox, A. Kalawa, and R. Williams, "The Implementation of a Dynamic Load Balancer," *Proceedings on Conference of Hypercube Multiprocessors*, pp. 114–121, 1987.

[15] L.V. Kale, "Comparing the Performance of Two Dynamic Load Distribution Methods," *Proceedings of the International Conference on Parallel Processing*, Vol I, pp. 8–12, 1988.

[16] P. Krueger and M. Livny, "The Diverse Objectives of Distributed Scheduling Policies," *Proceedings of the International Conference on Distributed Computing Systems*, pp. 242–249, 1987.

[17] F.C.H. Lin and R.M. Keller, "The Gradient Model Load Balancing Method," *IEEE Transactions on Software Engineering*, SE-13, pp. 32–38, 1987.

[18] R. Luling, B. Monien, and F. Ramme, "Load Balancing in Large Networks, A Comparative Study," *Proceedings of the Symposium on Parallel and Distributed Processing*, Dallas TX, pp. 686–689, 1991.

[19] Message Passing Interface (MPI) Standard. URL http://www.mcs.anl.gov/mpi/index.html.

[20] L. Oliker and R. Biswas, "Efficient Load Balancing and Data Remapping for Adaptive Grid Calculations," *Proceedings of the Symposium on Parallel Algorithms and Architectures*, Newport, RI, Jun. 1997, to appear.

[21] S. Pulidas, D. Towsley, and J. A. Stankovic, "Embedding Gradient Estimators in Load Balancing Algorithms," *Proceedings of the International Conference on Distributed Computing Systems*, pp. 482–490, 1988.

[22] C.G. Rommel, "The Probability of Load Balancing Success in a Homogeneous Network," *IEEE Transactions on Software Engineering*, pp. 922–923, Sept. 1992.

[23] V. Sarkar and J. Hennessy, "Compile-time Partitioning and Scheduling of Parallel Programs," *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press, Los Alamitos, CA, pp. 61–70, 1995.

[24] K.G. Shin and Y.C. Chang, "Load Sharing in Hypercube Multicomputers for Real-Time Applications," *Proceedings of the Conference on Hypercubes, Concurrent Computers, and Applications*, pp. 617–621, Vol. 1, 1989.

[25] B.A. Shirazi, A.R. Hurson, and K.M. Kavi, *Scheduling and Load Balancing in Parallel and Distributed Systems*, IEEE Computer Society Press, Los Alamitos, CA, 1995.

[26] N.G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *Computer*, pp. 33–44, December 1992.

[27] R.C. Strawn, R. Biswas, and M. Garceau, "Unstructured Adaptive Mesh Computations of Rotorcraft High-Speed Impulsive Noise," *Journal of Aircraft*, Vol. 32, No. 4, pp. 754–760, Jul./Aug. 1995.