Adaptive Load-Balancing Algorithms using Symmetric Broadcast Networks: Performance Study on an IBM SP2

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 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 existing techniques.*

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 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 [12] 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 [1, 5] allocate/reallocate resources at run time based on a set of system parameters that are maintained. Determining these parameters and how to broadcast them are important considerations.

In this paper, we consider general-purpose distributed memory parallel computers where 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, and processor workload is determined by the length of the local job queue of a node. The network is assumed to be homogeneous and any job can be processed by any node. However, jobs cannot be rerouted once execution begins.

Das et al. [3, 4] 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 a hypercube architecture. Based on [13], our SBN-based algorithms can be classified as:

Adaptive: performance adapts to the system load;
Symmetric: senders and receivers initiate balancing;
Stable: excessive balancing traffic is avoided;
Effective: balancing does not degrade performance.

The three algorithms proposed in this paper have been implemented on an IBM SP2 using the Message-Passing Interface (MPI). Performance of the SBN algorithms are analyzed by an extensive set of experiments with Poisson-distributed synthetic loads. The results are compared with other existing techniques such as Random [5], Gradient [9], Sender Initiated [6], Receiver Initiated [6], and Adaptive Contracting [6]. 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. Additional experiments where the SBN-based load balancing is applied to dynamic mesh adaptation problems using actual load data further confirm these conclusions [2].

This paper is organized as follows. Section 2 reviews a few existing 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 while Section 6 summarizes the experimental results, comparing SBN algorithms to other load-balancing techniques. The final section concludes the paper.

2 Previous Work

Among various approaches 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 [11], the probability of load-balancing success is computed analytically. In [7], several load-balancing methods are compared by implementing Fibonacci number generation, the N-Queens problem, and the 15-puzzle on a network; whereas the simulation approach has been employed in [8].

In this paper we perform experiments on an IBM SP2 multiprocessor, using the simulation approach with synthetically-generated random loads according to Poisson distributions. A mathematical analysis of the alorithms and results of additional experiments that utilize actual load data from a dynamic mesh application are reported in [2].

In general, load-balancing algorithms are very susceptible to the choice of system thresholds [10]. We have also noticed that proper selection of threshold values optimizes the proposed SBN-based algorithms. The following load-balancing algorithms will be compared with ours.

Random [5]: Jobs are randomly distributed among processors (or nodes) as they are generated. Once a job originating at a node is received by another node, it is processed.

Gradient [9]: Jobs migrate from overloaded to lightly-loaded nodes. This is accomplished by a maintaining a gradient at all nodes of the network. The gradient specifies the distance of the nearest lightly-loaded node through each neighbor. This requires frequent broadcasts between neighboring nodes.

Receiver initiated [6]: Load balancing is triggered by a lightly-loaded node. If the load value of a node falls 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. To prevent instability in light system load conditions, a node waits one second before initiating additional requests.

Sender initiated [6]: Messages are directed to lightly-loaded neighbors from overloaded nodes. To prevent instability under heavy system loads, nodes exchange load information with their neighbors when local job queue sizes are halved or doubled.

Adaptive contracting [6]: When jobs are generated, the originating node distributes bids to its neighbor nodes in parallel. The neighbors in turn respond with a message containing the number of jobs in their respective local queues. The originating node then appropriately distributes jobs to its neighbors.

3 Preliminaries on SBN

A *symmetric broadcast network* (SBN) defines a communication pattern (logical or physical) among the P processors in a multicomputer system [3, 4]. 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 a node are uniquely defined by the message originating node and the communication stage.

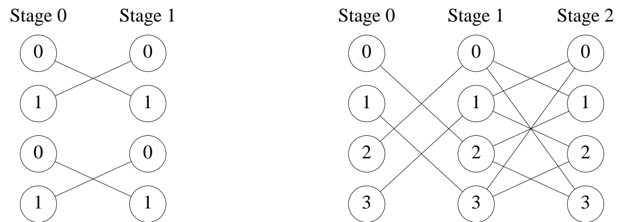

Figure 1: Construction of SBN(2) from two SBN(1)s

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) are for messages originating from node 5. Now if n_5^s denotes a node at stage

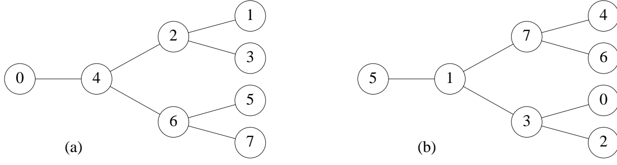

Figure 2: SBN communication patterns in SBN(3)

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, a *modified binomial spanning tree*, which is two binomial trees connected back to back, can be obtained. Figure 3 shows such a communication pattern for a 16-node SBN network which routes messages from node 0. The solid lines represent the actual SBN pattern, whereas the dashed lines are used to gather load-balancing messages at the destination node 15.

The modified binomial spanning tree is particularly suitable for adapting an SBN-based 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. 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.

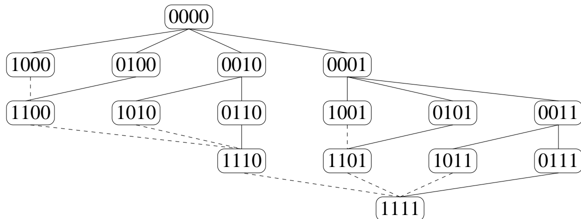

Figure 3: Binomial spanning tree in SBN(4)

4 Load Balancing Characteristics

4.1 System Thresholds

All SBN-based load-balancing 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 balance load. If `QLen` > `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 to receive 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 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 distribution messages. 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 SBN-based load-balancing algorithms consist of four key procedures. The first two, *Get-*

Distribute 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 algorithm 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 receive load-balance messages
  Call *GetDistribute* to receive distribution messages
  If (QLen > MaxTh)
    Call *Distribute* to send excess jobs through SBN
  If (QLen < MinTh)
    Call *Balance* to initiate load-balancing operation
    Call *UpdateLoad*(TotalJobsQueued) to set SysLL
  Normal Processing
End Repeat

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

Figure 4: Pseudo code for all SBN algorithms

5 Proposed algorithms

5.1 Standard SBN Algorithm

In the standard SBN algorithm, load-balancing messages are routed through the 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 < 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.

For illustration, consider SBN(3) in Fig. 6(a) which depicts the id and QLen for each node. For example, node 6 has three jobs queued for processing, indicated as $Q3$. The initial values of the SysLL, MinTh, and

Procedure *GetBalance*
While there are balance messages to receive
  Route needed jobs to predecessor node if possible
  If balancing messages are 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 load balancing
    Decrement number of active balance operations
    Call *UpdateLoad*(TotalJobsQueued)
    Distribute excess jobs and SysLL through SBN
  else
    Increment load-balancing operations in process
    Route the balance message to the next SBN stage
End While
Return

Procedure *GetDistribute*
While there are distribution messages to receive
  Enqueue any jobs received
  If predecessor node needs jobs, route excess jobs
  If load balancing is complete
    Decrement number of active balancing operations
    Call *UpdateLoad*(TotalJobsQueued)
  If this message completes a distribution
    If (QLen > MaxTh)
      Trigger load balancing
  else
    Call *Distribute* to route excess jobs through SBN
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 number of active balance operations
  Compute number of distribution messages expected
Compute 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 a normal distribution and load balancing is active
  Inhibit the distribution and **Return**
Compute number of excess jobs and jobs needed
Dequeue the jobs to be distributed
Distribute jobs and forward SysLL data to successors
Return

Figure 5: Standard SBN algorithm pseudo code

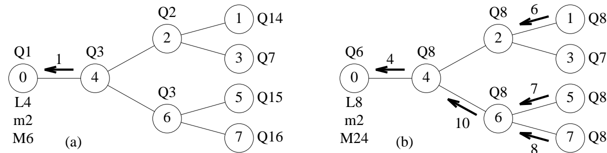

Figure 6: Standard SBN algorithm load-balancing

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:

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

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

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

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 predecessors when appropriate. Figure 6(b) shows the result of this distribution. The arrows indicate the number of jobs routed between nodes.

To balance loads of 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 Fig. 3. 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. 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, a load-balancing operation requires $3P - 4$ messages to be processed in $O(\log P)$ time.
- 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:

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

If $\texttt{ExJobs} = 0$ or if $\texttt{SysLL} > 2$ when $\texttt{ExJobs} = 1$, the balance message is forwarded to the next stage. Otherwise, the load balancing is terminated. The justification for this strategy is discussed in [2].

Job distribution is also processed differently in the heuristic SBN algorithm. For example, consider the network SBN(3) that has a processor with $\texttt{MaxTh} = 15$ and $\texttt{QLen} = 24$. 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 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 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 dis-

tribute messages more evenly, enhance network reliability, and allow various applications to be written using different communication patterns.

```
Procedure GetBalance
While there are balance messages to be processed
   Calculate the estimated TotalJobsQueued
   Call UpdateLoad(TotalJobsQueued)
   Distribute excess jobs to predecessor node
   If jobs distributed = 0 (or one job 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^(d−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 adjacent SBN nodes
Return(NumberOfJobsDistributed)
```

Figure 7: Heuristic SBN algorithm pseudo code

6 Experimental Results

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. Two 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.

Light system load (cf. Fig. 9): 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.

An additional experiment in which the system load transitions from heavy to light is reported in [2].

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.

The line charts included in Figs. 8-9 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 sent by a node.

(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 processing difference 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.

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

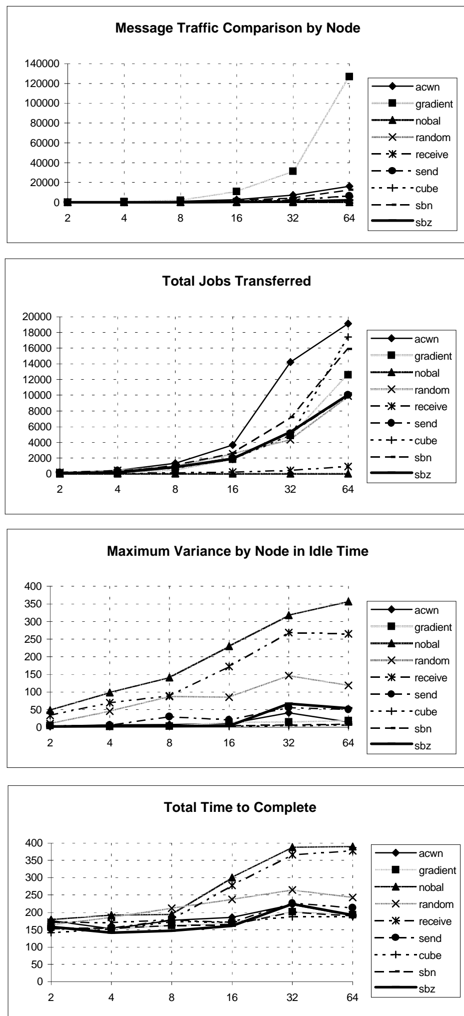

Figure 8: Heavy system load

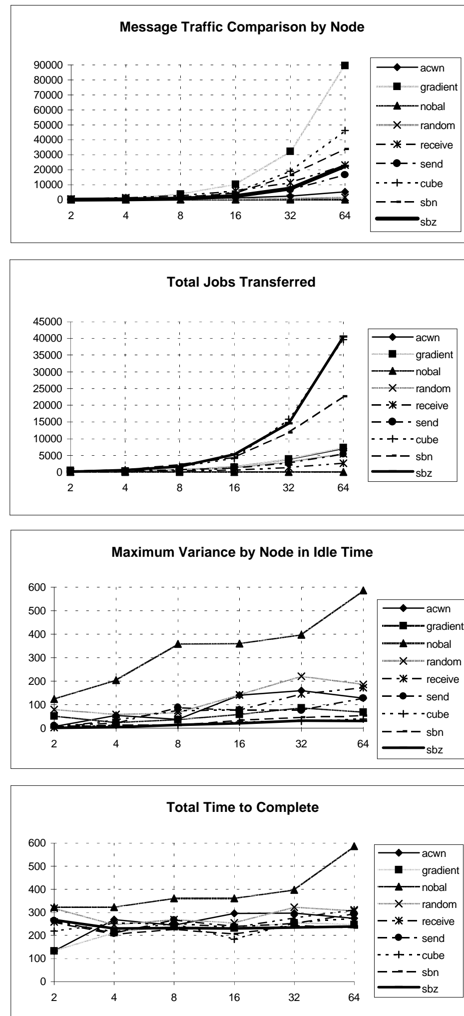

Figure 9: Light system load

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 significant and often results in excessive network traffic. The Adaptive Contracting (*acwn*) algorithm performs the best in periods of heavy system loads. However, as 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 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 proposed algorithms that we propose successfully balance the system load and minimize processor idle time. In addition, the heuristic variant reduces the overhead associated with balancing message traffic.

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. Another important topic 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. In the dynamic mesh adaptation reported in [2], 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] G. Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 7, No. 2, pp. 279–301, Oct. 1989.

[2] S.K. Das, D.J. Harvey, and R. Biswas, "Adaptive Load-Balancing Algorithms Using Symmetric Broadcast Networks," *NASA Ames Research Center Technical Report*, NAS-97-014, May 1997.

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

[4] S.K. Das, S.K. Prasad, C-Q. Yang, N.M.Leung, "Symmetric Broadcast Networks for Implementing Global Task Queues and Load Balancing in a Multiprocessor Environment," *UNT Technical Report*, CRPDC-92-1, 1992.

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

[6] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing," *Performance Evaluation*, Vol. 6, No. 1, pp. 53–68, Mar. 1986.

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

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

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

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

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

[12] 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.

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