Adaptive Load-Balancing Algorithms using Symmetric Broadcast Networks

Sajal K. Das

Center for Research in Wireless Mobility and Networking (CReWMaN)
Department of Computer Science and Engineering
The University of Texas at Arlington, Arlington, Texas 76019
E-mail: das@cse.uta.edu

Daniel J. Harvey

Department of Computer Science
Southern Oregon University, Ashland, Oregon 97520
E-mail: harveyd@sou.edu

Rupak Biswas

NASA Advanced Supercomputing (NAS) Division
NASA Ames Research Center, Moffett Field, California 94035
E-mail: rbiswas@nas.nasa.gov

Proposed running head:
ADAPTIVE LOAD BALANCING USING SBN

Address for manuscript correspondence:
Rupak Biswas
Mail Stop T27A-1
NASA Ames Research Center
Moffett Field, CA 94035-1000
E-mail: rbiswas@nas.nasa.gov
Phone: (650) 604-4411
Phone: (650) 604-3957

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 efficient SBN-based dynamic load-balancing algorithms, and implement them on an SGI Origin2000. A thorough experimental study with Poisson-distributed synthetic loads demonstrates that our algorithms are effective in balancing system load. By optimizing completion time and idle time, the proposed algorithms are shown to compare favorably with several existing approaches.

Key words: Dynamic load balancing; network topology; job migration; performance study.

1 Introduction

To maximize the performance of a multicomputer system, it is essential to evenly distribute the workload among the available processors. In other words, it is desirable to prevent, if possible, the condition where one processor is overloaded with a backlog of jobs to be serviced while another processor is lightly loaded or even idle. The load-balancing problem is closely related to scheduling and resource allocation, and can be either static or dynamic. In *static* load balancing algorithms [18], the decisions are made at compile time when resource requirements are estimated. On the other hand, *dynamic* algorithms [2, 7, 11, 12, 18, 23] allocate/reallocate resources at runtime based on a set of system parameters, which may determine when jobs can be migrated and also account for the associated overhead in such a transfer [17]. Determining the parameters to be maintained and how to broadcast them among processors are important design considerations, normally resolved by distributed scheduling policies [10, 13].

This paper deals with *decentralized* load balancing in distributed-memory multicomputers in which processors are connected by a point-to-point network topology and communicate with one another via message passing. The network is assumed to be homogeneous and any job can be serviced by any processor; however, jobs cannot be rerouted once execution begins. We have confined the scope of this paper to independent tasks only. The workload of a processor is determined by the length of its local job queue.

In particular, we propose three efficient dynamic load-balancing algorithms which make use of a logical and topology-independent communication pattern among processors, called a *symmetric broadcast network* (SBN), introduced in [5, 6]. These algorithms will be called (i) Basic SBN algorithm, (ii) Hypercube Variant, and (iii) Heuristic Variant.

SBN-based load balancing can be initiated by any processor that has too many or too few jobs to process based on certain threshold values. *Balancing messages* are first broadcast so that the current system load of jobs can be estimated. This is followed by *distribution messages* which reassign jobs to minimize the possibility of processors becoming idle. SBN-based load balancing runs concurrently while application processing continues. A topology-independent logical network, such as an SBN, helps provide predictable communication patterns to applications that make use of wide area networks of processors for both load balancing and interprocessor communication. An SBN is also effective when implementing message-passing applications for multicomputer systems in a portable manner.

The SBN topology can easily be embedded into specific networks if efficiency is an issue. For example, one of our SBN-based algorithms (Hypercube Variant) naturally adapts to a hypercube topology, and is thus used to compare with other hypercube-based dynamic load-balancing methods. This helps us determine whether the measured performance improvements are due to the effects of the network topology or the proposed load-balancing schemes themselves.

Based on their operational characteristics, our SBN-based algorithms can be classified (according to [19]) as: (a) *Adaptive*, since the performance adapts to the average number of queued jobs; (b) *Symmetrically Initiated*, since both senders and receivers can initiate load balancing; (c) *Stable*, since the network is not burdened with excessive load-balancing traffic; and (d) *Effective*, since system performance does not degrade while balancing workloads.

The performance of the proposed SBN-based algorithms is analyzed by conducting a thorough experimental study on a 32-processor SGI Origin2000 machine, using the Message-Passing Interface (MPI) paradigm. A preliminary version of this work that describes experiments running on an IBM SP2 is available in [3]. Investigating the programming paradigm is beyond the scope of this paper. Since the SBN strategy is topology and architecture independent, it made sense to use a portable library like MPI. Furthermore, any benefit provided by exploiting the Origin2000 shared-memory architecture would be equally available to all the other load-balancing schemes.

We use Poisson-distributed synthetic loads and compare the performance with other methods such as Random [7], Gradient [14, 15], Sender Initiated [8], Receiver Initiated [8], Adaptive Contracting [8], and Tree Walking [20], as summarized in Section 2. Our experiments demonstrate that the quality of load balancing achieved by the SBN approach compares favorably with respect to four metrics: (i) message traffic per processor, (ii) total jobs transferred, (iii) maximum variance in processor idle time, and (iv) total completion time. For example, under heavy system loads, the SBN algorithms complete in 6% to 22% less time than other balancing algorithms that are analyzed in this paper. Idle time is also reduced by over 67%. Under light system loads, the SBN-based algorithms incur significantly less message traffic as compared to other popular balancing algorithms such as Gradient and Receiver Initiated.

This paper is organized as follows. Section 2 reviews several existing approaches to dynamic load balancing that are used as comparisons. Section 3 defines SBNs, presents some of their properties, and discusses the general characteristics of our proposed load-balancing algorithms.

Section 4 presents three SBN-based load-balancing schemes and analyzes their performance characteristics. Section 5 contains experimental results that compare the SBN algorithms to other load-balancing techniques. The final section concludes the paper.

2 Related Work

A wide variety of dynamic load-balancing algorithms have been proposed for improving multiprocessor performance [2, 7, 8, 9, 11, 12, 14, 15, 18, 20, 21, 22, 23, 24]. Let us first summarize the underlying characteristics of some of the most popular methods which are used to compare the performance of our SBN-based algorithms.

Random [7]: If the number of jobs queued at a given processor is larger than a certain threshold, additional jobs are randomly distributed among its neighbors. Although a single distribution message may contain several jobs, a particular job cannot be migrated multiple times. In other words, once a job is migrated, it is queued for processing.

Gradient [14, 15]: Jobs migrate from overloaded to lightly-loaded processors based on a systemwide gradient. Each processor maintains, for each of its immediate neighbors, the minimum number of communication hops to the nearest lightly-loaded processor. Whenever these values change, they are broadcast to all the neighbors. However, because of network dynamics, this is only an approximation to the true system load. Each processor also has a load status flag which, by comparison with system thresholds, determines whether the processor is overloaded, lightly loaded, or moderately loaded. Jobs are routed to the neighbor lying on the path to the nearest lightly-loaded processor, and a job can migrate several times before being finally processed.

Receiver Initiated [8]: Load balancing is triggered by a processor with load level below the system threshold. The lightly-loaded processor broadcasts to its neighbors a job request message which contains information about its queue. Upon receiving this message, each neighbor compares its own queue length to that of the requesting processor. If the local queue size is larger, the neighboring processor replies with a single job. To prevent instability under light system load conditions, the requesting processor waits for a specified amount of time for a reply before initiating another job request. Like the Gradient scheme, it is possible for a job to be migrated multiple times before being finally processed.

Sender Initiated [8]: Unlike Receiver Initiated, load balancing occurs when processors become overloaded. To prevent instability under heavy system loads, each processor exchanges load information with its neighbors. More precisely, load values are exchanged when a local queue length is halved or doubled, so job migrations occur less frequently than system load changes. Additional jobs are distributed to lightly-loaded neighbors. Like Random, multiple job migration is not allowed.

Adaptive Contracting [8]: When jobs arrive at a processor, it distributes bids to all of its neighbors. The neighbors respond to the bid with a message containing the number of jobs in their respective local queues. The originating processor then distributes jobs to those neighbors that have loads smaller than the system threshold. The number of jobs migrated is such that jobs are equally distributed among the originating processor and its lightly-loaded neighbors.

Tree Walking [20]: Utilizing a binary tree topology, the fixed root processor initiates a load-balancing operation when one of the processors becomes idle. Processing is temporarily suspended when load balancing is underway. First, a balance message is broadcast through the network. Processors respond by sending their current load level back towards the root using a global reduction operation. The correct systemwide load level is then broadcast. Finally, jobs are distributed so that every processor has an equal number of jobs.

Despite some similarities between our SBN-based approach (details given later in Section 4) and the Tree Walking Algorithm (TWA), there are several major differences as outlined below:

- TWA always initiates load balancing from a single root processor. It is, therefore, more restrictive and has less potential for being fault tolerant than SBN, which allows any processor to initiate load balancing.
- Application processing is temporarily suspended when TWA executes because it is unable to mask balancing overhead by overlapping processing and load balancing. In contrast, SBN balancing proceeds concurrently with application processing.
- TWA is designed to perfectly equalize the job count at every processor leading to more message passing. SBN balancing attempts to minimize the possibility that a particular processor will become idle; thus, a perfectly equalized job count is unnecessary.
- TWA triggers load balancing only when processors become idle. SBN anticipates idle conditions and triggers balancing ahead of time.
- TWA requires communication messages to be broadcast to all processors when balancing the system. The SBN Heuristic Variant allows balancing to be accomplished without this requirement.

All of these algorithms can be classified as being *iterative* in nature because they strive to reach a global balanced state through successive nearest-neighbor operations by individual processors. Iterative methods, in general, are a good match for direct point-to-point interconnection networks commonly used for communications in modern multicomputers. Furthermore, they are flexible, preserve communication locality, and are inherently scalable.

There are two main classes of iterative methods: diffusion [2, 21, 23] and dimension exchange [2, 23, 24]. Diffusion algorithms require processors to communicate simultaneously with all of its nearest neighbors to reach a local load balance. On the other hand, in dimension exchange, a processor balances workload with each neighbor at a time. Diffusion algorithms are more popular because of their simplicity; however, their efficiency depends on a parameter that determines the behavior of the local balancing operation. In [12], an improved diffusion algorithm based on Chebyshev polynomials was proposed. Results showed that performance was better than the baseline diffusion method, but at the additional cost of calculating two eigenvalues. The drawback of the dimension exchange method is that equidistribution of the workload between a pair of processors at each balance operation is not necessarily efficient. It has therefore been generalized by using an exchange parameter to control the workload distribution between pairs of processors [22]. For processor networks that can be represented as a cartesian product of graphs, [9] proposed an alternating-direction dimension exchange scheme that reduces the number of iterations but at the cost of significantly greater job transfers. A refined scheme switches directions every other iteration to lower the amount of job migration.

3 Preliminaries

In this section, we define the concept of *symmetric broadcast network* (SBN), describe its properties, and present the general characteristics of our proposed load-balancing algorithms based on SBNs.

3.1 Definition of SBN

An SBN defines a communication pattern (logical or physical) among the P processors in a multicomputer system [5, 6]. An SBN(d) of dimension $d \geq 0$, is a $(d+1)$-stage interconnection network with $P = 2^d$ processors in each stage. It is constructed recursively as follows:

- A single processor forms the basis network SBN(0) consisting of a single stage, denoted as stage 0, with no communication link.
- For $d > 0$, an SBN(d) is obtained from a pair of SBN($d-1$)s as follows:
  (a) Keep the processor labels in the first SBN($d-1$) unchanged as 0 through $2^{d-1}-1$; relabel the processors of the second SBN($d-1$) as 2^{d-1} through $2^d - 1$.
  (b) Create an additional communication stage d, containing processors 0 through $2^d - 1$. Connect each processor j in stage $d-1$ to processor $(j + 2^{d-1}) \bmod 2^d$ in stage d.

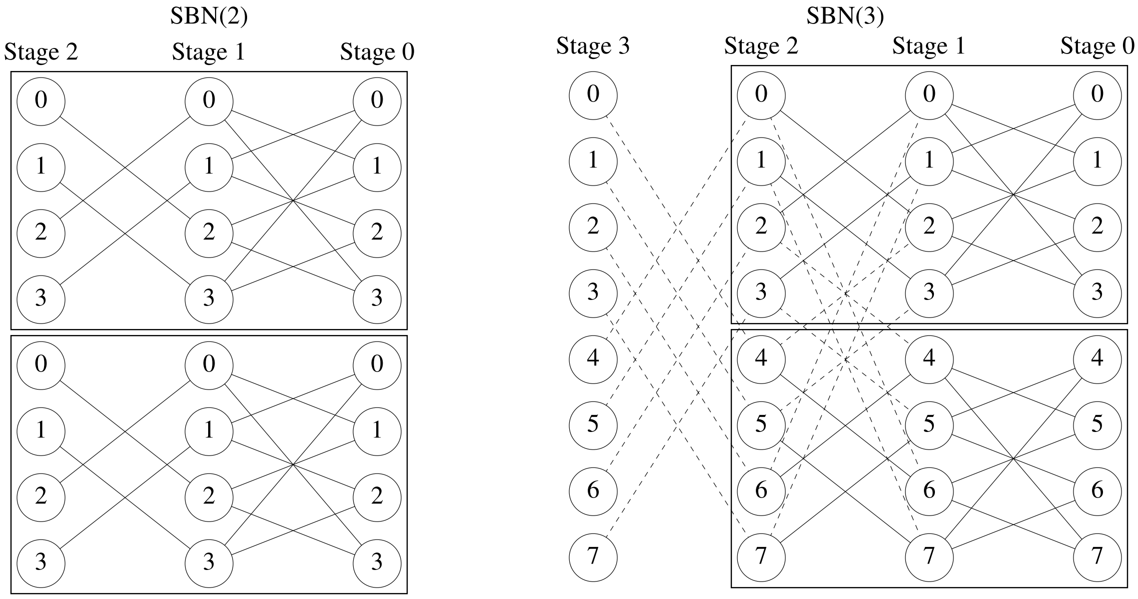


Figure 1: Construction of SBN(3) from a pair of SBN(2)s. Dashes indicate the new connections.

(c) If stage $d-2$ exists, for each processor j in stage $d-2$, define k to be the successor of j in stage $d-1$. Likewise, define m to be the successor of k in stage d (as just created in step (b) above). Connect processor j in stage $d-2$ to processor m in stage $d-1$.

An example of how an SBN(3) is formed from two SBN(2)s is shown in Fig. 1. An SBN defines unique communication patterns among the processors in the network. For any source processor at stage d of SBN(d), there are $d = \log P$ stages of communication where each processor appears exactly once. The successors and predecessors of each processor in a given stage i are uniquely defined by specifying the label of the originating processor and the communication stage number. Messages originating from source processors are appropriately routed through the SBN.

3.2 Properties of SBN

Among a total of eight possible communication patterns in SBN(3), consider the two patterns shown in Fig. 2. The paths in Fig. 2(a) are used to route messages originating from processor 0, while those in Fig. 2(b) are for messages originating from processor 5. Let n_5^s denote a processor label at stage s in Fig. 2(b) and let n_0^s be the corresponding processor label 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 processor in the communication pattern for messages originating from source processor x, then $n_x^s = n_0^s \oplus x$. Thus, all SBN communication patterns can be derived from the template pattern having processor 0 as the root. The predecessor and two successors of n_0^s can then be computed as follows:

Predecessor: $((n_0^s - 2^s) \vee 2^{s+1}) \bmod 2^d$, if $0 \leq s < d$ ($\vee$ is the inclusive-OR operator),

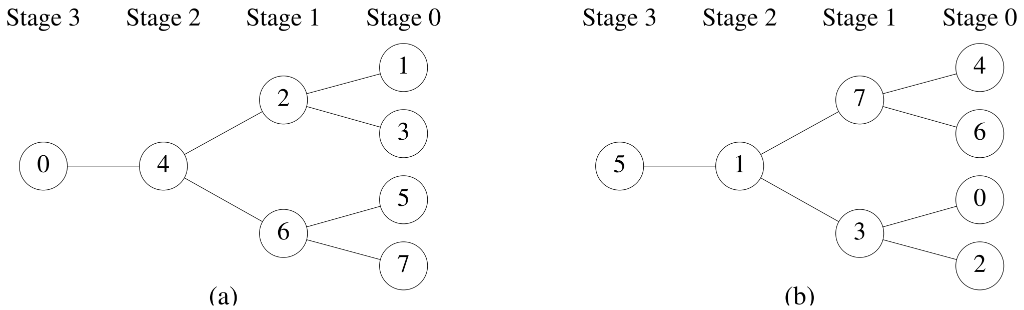


Figure 2: Two communication patterns in SBN(3).

Successor_1: $n_0^s + 2^{s-1}$, if $1 \leq s \leq d$,
Successor_2: $n_0^s - 2^{s-1}$, if $1 \leq s < d$.

Figure 2 illustrates two possible SBN communication patterns, but many others can easily be derived by slightly altering the SBN definition to match a given network topology and application requirements. Multiple randomly-selected SBN patterns help distribute messages more evenly, enhance network reliability, and allow various applications to be written using different communication patterns. For example, the SBN communication pattern for processor 0 can be defined with the help of a one-dimensional array implementation of a full binary tree such that the predecessor and successors of a processor are given by:

Predecessor: $\left\lfloor \frac{n_0^s}{2} \right\rfloor$, if $0 \leq s < d$,
Successor_1: $2 \times n_0^s + 1$, if $1 \leq s \leq d$,
Successor_2: $2 \times n_0^s$, if $1 \leq s < d$.

Let us demonstrate how to embed a hypercube topology within an SBN, as will be required for the Hypercube Variant of our load-balancing algorithm. This embedding uses a *modified binomial spanning tree*, which consists of two binomial trees[1] connected back-to-back. Figure 3 shows such a communication pattern for a 16-processor network which is used to route messages originating from processor 0. The solid lines of the diagram represent the actual SBN(4) pattern, whereas the dashed lines are used to gather load-balancing messages at a single destination processor (processor 15, in this case). This embedding ensures that all successors and predecessors at any communication stage are adjacent processors in the hypercube. Also, every originating processor has a unique destination. Finally, if the processors are numbered using a binary string of d bits, the number of predecessors for a processor x is $\max\{1, b\}$, where b is the number of consecutive leftmost 1-bits in the binary address of x.

[1]A binomial tree $B(k)$ of 2^k nodes is an ordered tree defined recursively as follows [1]: $B(0)$ consists of a single node; $B(k)$ consists of two $B(k-1)$s linked together such that the root of one is the leftmost child of the root of the other.

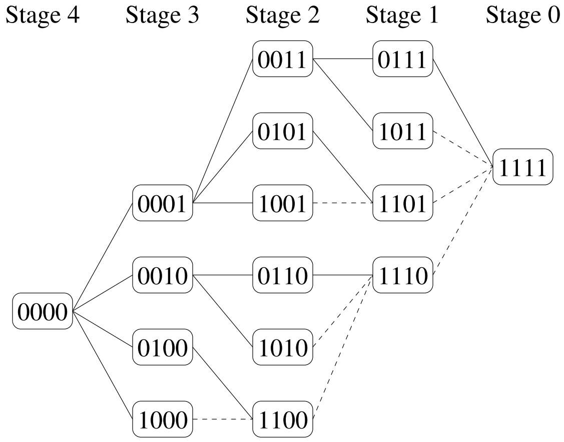


Figure 3: Modified binomial spanning tree used as a hypercube SBN(4).

3.3 General Characteristics of Proposed Load Balancers

We discuss below some general features that are shared by all three SBN-based adaptive load-balancing algorithms presented in Section 4. We also describe various system thresholds, the two types of messages that are processed by the SBN approach, and pseudo codes for the procedures common to all three algorithms.

3.3.1 System Thresholds

In general, many load-balancing algorithms are very susceptible to the choice of system thresholds [16]. A proper selection of these threshold values has proven helpful in optimizing our algorithms as well.

The proposed SBN-based algorithms adapt their behavior to the system load. Under heavy (respectively, light) loads, the balancing activity is primarily initiated by processors that are lightly (respectively, heavily) loaded. This activity is controlled by two *system thresholds*: `MinTh` and `MaxTh`, which are the minimum and the maximum system load levels. The *system load level*, `SysLL`, is the average number of jobs queued per processor. If a processor p has a queue length $\texttt{QLen}(p) < \texttt{MinTh}$, load balancing is initiated. If, on the other hand, $\texttt{QLen}(p) > \texttt{MaxTh}$, the extra jobs $\texttt{ExLoad} = \texttt{QLen}(p) - \texttt{MaxTh}$ are distributed through the network, without explicit load balancing. However, if this distribution overloads a processor in the final stage (stage 0), load balancing is triggered.

The performance of our algorithms is affected by the chosen values for `MinTh` and `MaxTh`. For instance, `MinTh` must be large enough to receive sufficient jobs before a lightly-loaded processor becomes idle. However, it should not be too large to initiate unnecessary load balancing. Similarly, 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. Basically, once there is sufficient load on the network, very little load-balancing activity should be required.

3.3.2 Message Types

Two types of messages are processed by the SBN-based algorithms. The first type is a *balancing message* which originates from an unbalanced processor and is then routed through the SBN. The cumulative total of queued jobs, `TotalJQ`, is computed to obtain a snapshot value of `SysLL`.

The second type of messages is a *distribution message*, which is used during a load-balancing operation to route the `TotalJQ` value though the network and to reassign jobs after the balancing message is broadcast. Each processor upon receipt of such a message, updates its local values of `SysLL`, `MinTh`, and `MaxTh`. Distribution messages can also be sent to predecessor processors prior to completing the broadcast of the balancing message. This action occurs so that jobs can be assigned to predecessors having less than `MinTh` jobs queued and about to go idle. Finally, distribution messages are sent when a load-balancing operation is not in progress and a processor has greater than `MaxTh` jobs queued. In this case, `ExLoad` jobs are reassigned.

If the communication from one processor 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 can be shown to be $O(\log^2 P)$ [5]. To reduce message traffic, a processor does not initiate additional load-balancing activity until all previous balancing messages passing through it have been serviced.

3.3.3 Common Procedures

All three of our load-balancing algorithms consist of four key procedures. The first two, *GetBalance* and *GetDistribute*, are used to process balancing and distribution messages that are received, while the other two, *Balance* and *Distribute*, route those messages to the successors in the SBN. Implementation details of these procedures depend on the particular load-balancing algorithm used. Figure 4 presents the pseudo code that is common to all our SBN-based load-balancing algorithms, and is executed in parallel by every processor. The *UpdateLoad* procedure adjusts the system thresholds described in Section 3.3.1. It is called by the *GetBalance* and *GetDistribute* procedures when load-balancing operations complete.

In our experiments, we found that setting `ConstParam` $= 2$ proved to be the most effective value. `MinTh` is then set so that load balancing will begin before processors go idle. The setting of `MaxTh` grows exponentially with `SysLL` because the need for load-balancing activity decreases rapidly as `SysLL` increases. The mathematical justification for this policy is presented in Section 4.4.

```
Procedure Main
Repeat forever
    Call GetBalance to process balancing messages received
    Call GetDistribute to process distribution messages received
    If (QLen(p) > MaxTh)
        ExLoad = QLen(p) − MaxTh
        Call Distribute to route ExLoad jobs through the SBN
    If (QLen(p) < MinTh)
        Call Balance to initiate a balance operation and determine TotalJQ
    Resume processing of application program
End Repeat

Procedure UpdateLoad(TotalJQ)
SysLL = ⌈ TotalJQ / P ⌉
MaxTh = SysLL + 2 ⌊ SysLL / ConstParam ⌋
MinTh = SysLL − 1
If (SysLL > ConstParam) MinTh = ConstParam
Return
```

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

4 SBN-Based Load-Balancing Schemes

Based on the SBN concept, a baseline dynamic load-balancing technique and two variants are proposed in this paper. The Basic algorithm and the Hypercube Variant attempt to accurately compute and maintain the value of `SysLL`, whereas the Heuristic Variant estimates it.

4.1 Basic SBN Algorithm

In this algorithm, balancing messages are routed through SBN(d) from the source (root) at stage d to all the processors at stage 0. These messages are then routed back to the root so that `TotalJQ` can be computed. Thus, the originating root processor has an accurate snapshot value of `SysLL`. Next, distribution messages are sent to relocate jobs and to broadcast the `TotalJQ` value. All processors then update their local `SysLL`, `MinTh`, and `MaxTh`. The extra load of jobs (`ExLoad`) are routed as part of this distribution to balance the system load. In addition, if $\texttt{QLen}(p) < \texttt{SysLL}$ for a processor p, the need for jobs is indicated during the distribution process. Successor processors respond by routing back an appropriate number of excess jobs, if available. Figure 5 presents the pseudo code of the Basic SBN algorithm where $\texttt{Stage}(p)$ denotes the SBN stage of processor p for a given communication pattern (cf. Fig. 2) and `JobsRecv` is the number of jobs received by it in a distribution message.

To illustrate the operation of this algorithm, consider SBN(3) in Fig. 6(a). The label and $\texttt{QLen}(p)$ for each p are shown inside and outside the corresponding circle. For example, proces-

```
Procedure GetBalance
While (balancing messages remain to be processed)
    If (QLen(processor sending this message) < MinTh)
        Route ⌊ QLen(p) / 2 ⌋ jobs to processor sending this message
    If (no balance operation active from the SBN root processor)
        Increment count of simultaneous balance operations being serviced
        If (Stage(p) == 0) Call Balance to route QLen(p) value towards the root processor
        Else
            Indicate two balancing messages to be gathered from successors
            Call Balance to route balancing message to successors
    Else
        If (second balancing message remains to be gathered from successors) Continue
        If (Stage(p) == d)
            Call UpdateLoad(TotalJQ)
            ExLoad = QLen(p) − SysLL
            Call Distribute to route ExLoad jobs and TotalJQ value to successors
            Decrement count of simultaneous balance operations being serviced
        Else Call Balance to route (QLen(p) + QLen(successors)) value to the root processor
End While
Return

Procedure GetDistribute
While (distribution messages remain to be processed)
    QLen(p) = QLen(p) + JobsRecv
    If (TotalJQ value contained in message received)
        Call UpdateLoad(TotalJQ)
        Route min (QLen(p) − SysLL, SysLL − QLen(predecessor)) jobs to predecessor
        Decrement count of simultaneous balance operations being serviced
    If (Stage(p) == 0)
        If (QLen(p) > MaxTh) Call Balance to initiate new balance operation
    Else
        ExLoad = QLen(p) − MaxTh
        If (this is a balance operation) ExLoad = QLen(p) − SysLL
        Call Distribute to route ExLoad jobs and/or TotalJQ value to successors
End While
Return

Procedure Balance
If (Stage(p) == d)
    If (balance operation already underway) Return
    Increment count of simultaneous balance operations being serviced
    Indicate that a balancing message is expected from successor
Route the balancing message to next SBN stage
Return

Procedure Distribute
If ((this is a non-balance distribution) And (balance operation already underway))
    Inhibit the distribution and Return
QLen(p) = QLen(p) − ExLoad
If ((ExLoad > 0) Or (TotalJQ value to send)) Route the distribution message to successors
Return
```

Figure 5: Pseudo code for the Basic SBN algorithm.

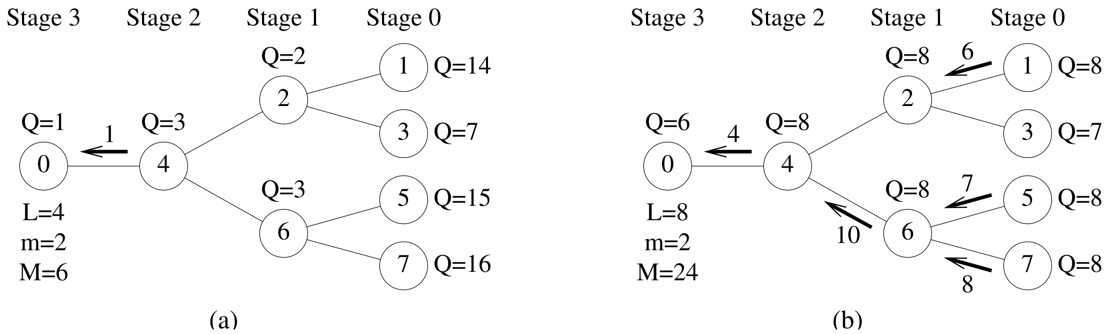


Figure 6: An example of load balancing using the Basic SBN algorithm: (a) during balancing messages, and (b) during distribution messages.

sor 6 has $Q = 3$ jobs queued for execution. At processor 0, the initial values are $\texttt{SysLL} = L = 4$, $\texttt{MinTh} = m = 2$, and $\texttt{MaxTh} = M = 6$. After a load-balancing request is sent through the SBN and then routed back to processor 0, these values are updated as $L = 8$, $m = 2$, and $M = 24$, using the *UpdateLoad* procedure given in Fig. 4. Note that when the balancing is initiated, processor 4 distributes half of its $\texttt{QLen}(p)$ jobs (i.e. $\lfloor 3/2 \rfloor = 1$) back to processor 0. This distribution is shown by the label on the arrow in Fig. 6(a).

Distribution messages are then used to route excess jobs to the successor processors or to indicate a need for jobs if $\texttt{QLen}(p) < \texttt{SysLL}$. Jobs are routed back to the predecessors when appropriate. Figure 6(b) shows the result of this distribution where the labels on arrows indicate the number of jobs routed between processors.

To balance P processors, $P - 1$ balancing messages are first sent through the SBN, which are then routed back to the root processor so that the SysLL value can be computed through a global reduction operation. Finally, $P - 1$ distribution messages are sent to balance the load as well as broadcast the TotalJQ value through the network. Note that, if a processor has an immediate need for jobs that can be supplied during load balancing, additional distribution messages are sent from neighboring processors to satisfy the need. If J such messages are required, a total of $3P - 3 + J$ messages will be have to be processed. Since $J = O(P)$, the total number of messages to be processed is $O(P)$. The depth of the SBN network being $O(\log P)$, the total time required to complete a load-balance operation is $O(\log P)$.

4.2 Hypercube Variant

The SBN approach has been adapted for implementation on a hypercube topology, using the modified binomial spanning tree as illustrated in Fig. 3. This algorithm uses the same control variables (SysLL, MinTh, and MaxTh), and processes balancing and distribution messages in the same manner as the Basic SBN algorithm. However, in embedding the SBN onto the hypercube, a processor r at stage d sends a balance message to each of its adjacent processors at stage $d - 1$ of the hypercube.

These messages are then routed through the hypercube network and eventually collected at the single destination processor q at stage 0. Processor q then accurately computes the current `SysLL` value and initiates the distribution process by routing distribution messages back towards the root processor at stage d. Note that the exclusive-OR property described in Section 3.2 still holds: it allows any processor to correctly determine the successors and predecessors if the stage number and the root processor are given.

Other differences between the Basic SBN algorithm and the Hypercube Variant are as follows:

- In the Hypercube Variant, the value of `TotalJQ` is computed when all balancing messages arrive at the destination processor. Unlike the Basic algorithm, this is possible because there is a unique destination for every originating processor in the hypercube embedding. Distribution messages are then routed back to complete the load balancing. Since there are $P - 1 + P/2 - 1$ interconnections in the modified binomial spanning tree (cf. Fig. 3), a load-balancing operation requires $3P - 4$ messages excluding the distribution messages sent between neighbors to satisfy the immediate need for jobs.

- Balancing messages always proceed from the root processor (at stage d of SBN in the Hypercube Variant) towards stage 0. This contrasts with the Basic SBN algorithm where balancing messages first proceed from stage d to stage 0, and are then routed back to stage d.

- To minimize the communication overhead in the Hypercube Variant, messages are gathered from the previous stage whenever more than one message is expected. The Basic SBN algorithm only needs to gather messages that are being routed back toward the root processor (because messages going in the other direction have only one predecessor).

- The network topology for the Hypercube Variant is such that the number of predecessor and successor processors vary at different communication stages, thereby somewhat complicating the implementation.

4.3 Heuristic Variant

Both the Basic algorithm and the Hypercube Variant are expensive since a large number of messages has to be processed to accurately maintain the `SysLL` value. The Heuristic Variant attempts to reduce this overhead by terminating load-balancing operations initiated by processor p as soon as `QLen`(p) is sufficiently large. In general, this strategy reduces the number of messages although $O(P)$ messages are still needed in the worst case. The pseudo code in Fig. 7 gives the operational details of the Heuristic Variant of the SBN-based load-balancing algorithm.

As in the Basic algorithm, p initiates load balancing when `QLen`(p) $<$ `MinTh` by sending a balancing message to its SBN successor. However, in the Heuristic Variant, the processor r that receives this message estimates `SysLL` by averaging local queue lengths over the processors through

```
Procedure GetBalance
While (balancing messages remain to be processed)
    TotalJQ = ⌈ P × (QLen(p) + QLen(predecessors)) / (d − Stage(p) + 1) ⌉
    OldSysLL = SysLL
    Call UpdateLoad(TotalJQ)
    ExLoad = ⌊ QLen(p) / 2 ⌋
    If (ExLoad > 0)
    If (QLen(p) ≤ SysLL)
        Call Balance to route balancing message to successors
        Else
        Call Distribute to route ExLoad jobs to predecessor
End While
Return

Procedure GetDistribute
While (distribution messages remain to be processed)
    JobsQueued = QLen(p)
    QLen(p) = QLen(p) + JobsRecv
    If (distribution towards the root processor)
        TotalJQ = P × (JobsQueued) + ⌈ JobsRecv / (d − Stage(p) + 1) ⌉ )
        Exload = ⌊ QLen(p) / 2 ⌋
    Else TotalJQ = P × (JobsQueued + ⌈ JobsRecv / (2^(Stage(p)+1) − 1) ⌉ )
        ExLoad = QLen(p) − SysLL
    Call UpdateLoad(TotalJQ)
    If (ExLoad > 0)
        Call Distribute to route ExLoad jobs to next SBN stage
End While
Return

Procedure Balance
If (Stage(p) == 0) Return
If (Stage(p) == d)
    TotalJQ = P × QLen(p)
    Call UpdateLoad(TotalJQ)
Route the balancing message to successors
Return

Procedure Distribute
If (Stage(p) == 0) Return
If (Stage(p) == d)
    TotalJQ = P × (SysLL + ⌈ QLen(p) − SysLL / P ⌉ )
    Call UpdateLoad(TotalJQ)
    ExLoad = QLen(p) − SysLL
QLen(p) = QLen(p) − ExLoad
Route ExLoad jobs with the distribution message to next SBN stage
Return
```

Figure 7: Pseudo code for the Heuristic Variant.

which the balancing message has already passed. If $\texttt{QLen}(r) > \texttt{SysLL}$, an appropriate number of jobs ($\texttt{ExLoad} = \lfloor \texttt{QLen}(r)/2 \rfloor$) is returned via a distribution message as shown in the *GetBalance* procedure in Fig. 7. In this case, the load-balancing procedure is also terminated. If instead, $\texttt{QLen}(r) \leq \texttt{SysLL}$, the balancing message is forwarded to the next SBN stage. Mathematical justifications for the Heuristic Variant are discussed in the next subsection.

Job distribution is also accomplished differently in the Heuristic Variant. If p determines that $\texttt{QLen}(p) > \texttt{MaxTh}$, it implies a job distribution is necessary. A new estimate of SysLL is calculated by p as $\texttt{SysLL} = \texttt{SysLL} + \lceil(\texttt{QLen}(p) - \texttt{SysLL}) \ / \ P\rceil$. It then tries to evenly distribute the excess load among all the processors in the network. Each processor r receiving a resulting distribution message with JobsRecv jobs updates its own SysLL and $\texttt{QLen}(r)$ values. However, before updating $\texttt{QLen}(r)$, its SysLL is computed as $\texttt{SysLL} = \texttt{QLen}(r) + \lceil \texttt{JobsRecv} \ / \ R\rceil$, where R is the number of processors (including r) in the remaining SBN stages. Note that SysLL is updated based on $\texttt{QLen}(r)$ and not its original outdated value. Processor r, in turn, distributes jobs in excess of its own updated SysLL to the next SBN stage. This is reflected in the formulae shown in the *GetDistribute* procedure in Fig. 7.

As an illustration, consider an SBN(3) that has a processor p with $\texttt{SysLL} = 7$, $\texttt{MaxTh} = 15$, and $\texttt{QLen}(p) = 24$. The newly-computed SysLL value is $7+\lceil(24-7)/8\rceil = 10$. The number of jobs that will be distributed to the successor (only one successor at this stage) is $\texttt{JobsRecv} = 24 - 10 = 14$. Suppose that the successor r has $\texttt{SysLL} = 9$, and $\texttt{QLen}(r) = 6$. After receiving 14 jobs from p, it has $\texttt{SysLL} = 6 + \lceil 14/7\rceil = 8$. Thus, $20 - 8 = 12$ excess jobs will be distributed to the next stage.

A significant advantage of the Heuristic Variant is that the balancing messages do not have to be gathered in order to calculate SysLL. This reduces the interdependencies associated with the communication and allows fault tolerance. If a particular processor fails, load balancing can still be accomplished with the help of the remaining processors.

An additional improvement for both the Basic SBN load-balancing algorithm and its Heuristic Variant can be obtained by using multiple communication patterns in the SBN. Each time a message is initiated, one of the SBN patterns is randomly chosen. Our experiments make use of the two communication patterns mentioned in Section 3.2 for computing predecessors and successors. Each of the balancing and distribution messages includes the source processor, the pattern used, and the stage to which the message is being routed. Since all processors have the SBN template associated with messages originating from processor 0, the required SBN communication pattern can be determined.

4.4 Analysis of SBN Message Passing

In a multicomputer consisting of P processors, we assume the arrival of jobs can be modeled by a Poisson distribution such that the probability of a processor having j jobs is $\lambda^j \ / \ (e^{\lambda} j!)$, where λ is the mean arrival rate. If $\texttt{SysLL} = k$, then by definition, the average number of jobs assigned to a processor is k. Hence, the probability that a processor has exactly j jobs is $k^j \ / \ (e^k j!)$. This implies the probability, g_j, that a processor in the network has more than j jobs is given by $g_j = 1 - (\sum_{i=0}^{j} k^i/i!) \ / \ e^k$. For example, 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 almost unity if $k > 15$. This implies that the need for load-balancing activity rapidly decreases as `SysLL` increases. Therefore, it makes sense to increase `MaxTh` exponentially with increasing `SysLL`.

To analyze the Heuristic Variant, let us compute the *expected number of jobs*, `EJobs`, that are returned to the processor that initiates the SBN load-balancing algorithm. We also compute the *expected number of processors*, `EProcs`, that will be visited during a balancing operation. In this way, we can determine whether sufficient jobs are returned and if message traffic is reduced by utilizing the Heuristic Variant. In the following, we mathematically model SBN message passing. For this purpose, we define the following four probability vectors:

- $\Phi = \langle \phi_0, \phi_1, \cdots, \phi_n \rangle$ defines the distribution of jobs queued at a given processor, where ϕ_i is the probability that the processor has i jobs queued for a specified value of `SysLL`.
- $\Phi_{stop} = \langle 0, 0, \cdots, 0, \phi_{stop}, \phi_{stop+1}, \cdots, \phi_n \rangle$ is the likelihood that balancing will terminate at a given processor. Here, $stop$ is the number of jobs queued at a processor that will prevent the SBN Heuristic Variant from forwarding a balancing message to the next stage.
- $\Phi_{continue} = \langle \phi_0, \phi_1, \cdots, \phi_{stop-1}, 0, \cdots, 0 \rangle$ defines the likelihood that balancing messages will be sent to successor processors.
- $C(V)$ is the probability vector computed by applying the function C to the probability vector V, and indicates the number of jobs distributed to a processor's predecessor. Here, V is the vector defining the probabilities of jobs queued at a processor after receiving distribution messages from its successors.

Proposition 1 *If* $V_1 = \langle v_{10}, v_{11}, \cdots, v_{1n} \rangle$ *and* $V_2 = \langle v_{20}, v_{21}, \cdots, v_{2m} \rangle$ *are probability vectors, then* $V_1 \bigotimes V_2 = \langle v_0, v_1, \cdots, v_{m+n} \rangle$*, where* $v_i = \sum_{j+k=i} v_{1j} v_{2k}$ *is also a probability vector.*

Proof. Consider the product $(\sum_{i=0}^{n} v_{1i})(\sum_{j=0}^{m} v_{2j})$. Since V_1 and V_2 are probability vectors, $\sum_{i=0}^{n} v_{1i} = \sum_{j=0}^{m} v_{2j} = 1$ and the above product is unity. This product can also be written as $v_{10}(v_{20} + v_{21} + \cdots + v_{2m}) + \cdots + v_{1n}(v_{20} + v_{21} + \cdots + v_{2m})$. All the terms in this Cartesian product can be reorganized as $(v_{10}v_{20}) + (v_{10}v_{21} + v_{11}v_{20}) + (v_{10}v_{22} + v_{11}v_{21} + v_{12}v_{20}) + \cdots$, where the expression within each pair of parenthesis is a vector entry of $V_1 \bigotimes V_2$. Since we have shown that this sum is unity, $V_1 \bigotimes V_2$ is also a probability vector. $\square$

Utilizing the above definitions, define $R_k = \langle j_{k0}, j_{k1}, \cdots, j_{kn} \rangle$ as a probability vector, where j_{ki} is the probability that i jobs from an SBN processor at stage k are returned to a predecessor at SBN stage $k+1$ during a balancing operation. Then, R_{d-1} indicates the jobs returned to the root processor that initiated the load-balancing operation, and can be computed by the following recursive formula:

Stage 0: $R_0 = C(\Phi)$, and
Stage $1 \leq k < d$: $R_k = C(\Phi_{stop} + \Phi_{continue} \bigotimes R_{k-1} \bigotimes R_{k-1})$,

where the function C reflects the SBN job return policy as charted in Fig. 8. Basically, the calculation function is used to add the jobs queued locally to the jobs that are expected to be returned from the two successor processors. Once $R_{d-1} = \langle r_0, r_1, \cdots r_n \rangle$ is computed, $\texttt{EJobs} = \sum_{j=0}^{n}(j \times r_j)$ can also be calculated.

To compute `EProcs`, let $\phi_c = \phi_0 + \phi_1 + \cdots + \phi_n$ be the probability that a given processor has $c > stop$ number of jobs queued. The probability that a balancing message will reach a processor at stage $k < d$ in the SBN is therefore ϕ_c^{d-k-1}, since the message must pass through $d-k-1$ predecessors. Since the number of processors at stage $k < d$ in SBN is 2^{d-k-1}, we have $\texttt{EProcs} = \sum_{k=0}^{d-1}(\phi_c^{d-k-1} \times 2^{d-k-1})$. For example, if $d = 5$ and $\phi_c = 0.4$, the expected number of processors visited during a balancing operation is 3.3616.

From the above model, we can compute `EJobs` and `EProcs` for various values of the system load level (`SysLL`), the dimension d of the SBN, and the value of $stop$. Assuming a Poisson job arrival rate, the probability vector Φ is easily obtained. The vector $C(V)$ can be calculated by utilizing the job return policy of the Heuristic Variant.

Figure 8 plots `EProcs` and `EJobs` values for this mathematical model. Here, $P = 32$ and `SysLL` varies between 1 and 15. In the graphs, we analyze five prospective policies for a processor to terminate load balancing, and compare their expected performance against the Basic SBN algorithm. Recall that when a processor p executes the Basic algorithm after receiving a balancing message, this message is forwarded to the next SBN stage unless p is at stage 0. The five policies for p to stop balancing are:

$\mathcal{P}_i$: when $\texttt{QLen}(p) \geq \texttt{SysLL} + 5 - i$ for $1 \leq i \leq 4$,
$\mathcal{P}_5$: when $\texttt{QLen}(p) \geq 2$ if $\texttt{SysLL} \leq 2$, or when $\texttt{QLen}(p) \geq 4$ for other `SysLL` values.

Notice that policy $\mathcal{P}_5$ attempts to return at least one job under light system loads and at least two jobs under heavier loads.

Figure 8(a) shows that if the Heuristic Variant is used with $stop = \texttt{SysLL} + 1$ (policy $\mathcal{P}_4$), `EProcs` is significantly less than that of the Basic SBN algorithm. For example, if $P = 32$ and $\texttt{SysLL} = 4$, only about 8 processors are visited on average using the Heuristic Variant, whereas all 31 processors are visited for the Basic algorithm. Furthermore, with the Heuristic Variant, Fig. 8(b) indicates that $\texttt{EJobs} = 4.2$ (compared to 8.8 with the Basic algorithm), a value much closer to the `SysLL` value of 4. Therefore, a better load balance is achieved with significantly reduced balancing traffic.

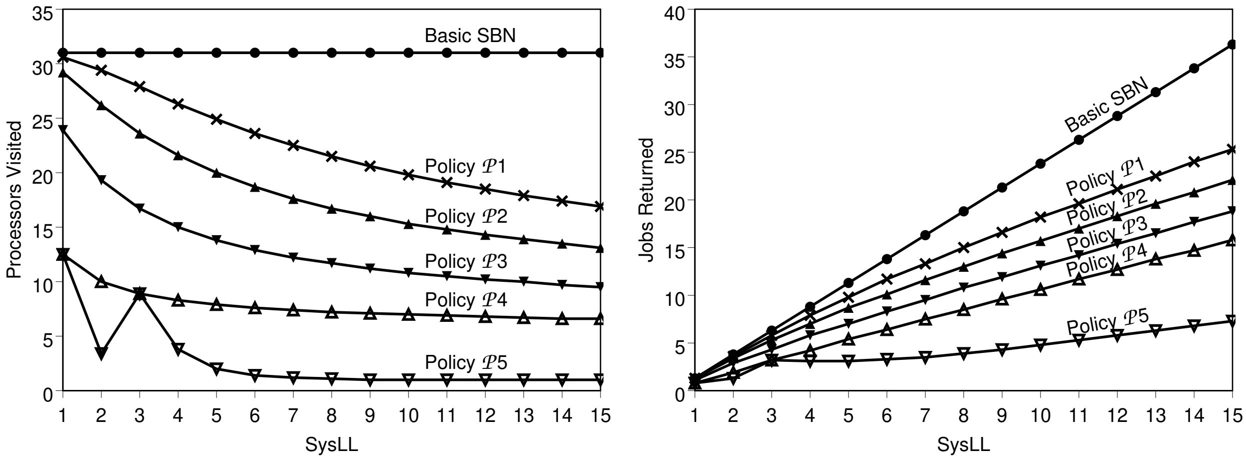


Figure 8: Expected number of (a) processors visited (`EProcs`), and (b) jobs returned (`EJobs`) for $P = 32$.

Figure 8(b) also gives an indication about optimal $stop$ values with respect to `SysLL`. After a successful load-balancing operation, $\texttt{QLen}(p)$ for each processor p should approximate the `SysLL` value. By the definition of SBN load-balancing algorithms, the processor that initiates load balancing has less than `MinTh` jobs queued for processing. `EJobs` will then be optimal when it is almost equal to `SysLL`. Figure 8(b) shows that this objective is achieved when $stop = \texttt{SysLL} + 1$. This is intuitively correct because it implies that a processor will forward a load-balancing message to the next SBN stage until an overloaded processor is encountered.

5 Experimental Study

This section describes the simulation environment and the test cases that were used to compare the proposed SBN-based load-balancing algorithms with several existing methods. The performance metrics used for comparison are explained, and comprehensive results obtained from simulation experiments are presented.

5.1 Simulation Environment

All the load-balancing algorithms were implemented using MPI and tested with synthetically-generated workloads on a 32-processor SGI Origin2000 located at NASA Ames Research Center. We used MPI rather than OpenMP to implement the various algorithms since investigating the programming paradigm is beyond the scope of this paper. The SBN strategy is topology and architecture independent; thus it made sense to use a portable library like MPI. Furthermore, any benefit provided by exploiting the Origin2000 shared-memory architecture would be equally available to all the load-balancing schemes.

The simulation program spawns the appropriate number of child processes and creates the desired SBN topology. A list of all process labels and an initial distribution of jobs are then routed through the network. In addition to the initial load, each processor dynamically generates additional jobs to be processed. Specifically, 10 job creation cycles are run where the number of jobs generated at each processor during a cycle follows a Poisson distribution. By randomly picking different values of λ (the mean arrival rate), varying numbers of jobs are created. Therefore, both heavy and light system load conditions are dynamically simulated. We have confined the scope of this paper to independent tasks only. Jobs are processed by "spinning" for the designated amount of time. The simulation terminates when all jobs have been processed. Note that as the number of processors increases, the workload also increases by the same factor. It is thus expected that the completion time should remain relatively constant if the algorithm scales efficiently.

Our experiments compare the performance of the SBN-based load-balancing algorithms with six other commonly-used techniques. They are Random, Gradient, Receiver Initiated, Sender Initiated, Adaptive Contracting, and Tree Walking, as summarized in Section 2. The Basic SBN scheme, its Heuristic Variant, and the Tree Walking algorithm utilize the SBN topology in their implementation. The SBN topology, being somewhat similar to the binary tree structure required by Tree Walking, provides a direct and fair comparison. The other algorithms and the SBN Hypercube Variant are implemented utilizing a hypercube topology. This also demonstrates the ability to embed the SBN approach onto another topology, and provides a fair comparison between the Hypercube Variant and other load-balancing methods. By comparing the Hypercube Variant and the Basic SBN algorithms, we can determine whether a change in topology results in any significant performance difference. Finally, the same experiments are also performed without any load balancing, in order to have a reference point. All ten algorithms (six existing load balancers, three proposed schemes, and no load balancing) are implemented using the same hardware and software environment.

The results obtained from these experiments are shown in Figs. 9, 10, and 11. They were compiled from repeating the simulations 10 times and averaging the results. We were limited to 10 runs by system usage limitations. Three separate load scenarios are considered as described below:

Heavy System Load (cf. Fig. 9): An initial load of 10 jobs per processor is queued during the first cycle of execution. Each execution cycle, including the first, is 1.0 sec in duration. At the start of the remaining nine cycles, an average of 19.41 additional jobs are generated in accordance with the formula $200\lambda^j \ / \ (e^\lambda j!)$, where λ and j are randomly varied between 1 and 10. The constant, 200, is large enough to ensure that the time required to process the created jobs is approximately double the 10.0 secs of execution (1.0 sec for each execution cycle), thereby guaranteeing an overloaded condition. The duration of all jobs average 0.1 sec, with the longest jobs requiring 0.2 sec. The entire simulation requires 10.0 secs plus the time needed to empty out all of the job queues. Optimal balancing for this experment requires an average of 1.0 sec to process the initial load, plus $19.41 \times 9 \times 0.1$ secs for processing the additional load cycles, amounting to 18.469 secs.

Transition from Heavy to Light System Load (cf. Fig. 10): An initial load of 50 jobs per processor is queued to a small subset ($\log P$) of processors during the first execution cycle. This ensures the initial heavy load condition. Each execution cycle, including the first, is 4.0 secs in duration. At the start of the remaining nine cycles, an average of 12.96 additional jobs are generated using the formula $260\lambda^j \,/\, (e^\lambda j!)$, where the values of λ and j are randomly varied between 1 and 20. The constant, 260, is chosen so that a light load of jobs will be generated at each execution cycle. The duration of all jobs average 0.2 sec, with the longest jobs requiring 0.4 sec. If load balancing is effective, the entire simulation requires 40.0 secs (4.0 secs for each execution cycle). Note that 10.0 secs is required to process the initial load, plus $12.96 \times 9 \times 0.2$ secs for processing the remaining cycles. This totals to 33.328 secs, leaving an average of 0.667 sec of idle time per cycle.

Light System Load (cf. Fig. 11): This experiment is similar to the previous one except that the initial load of jobs is very light. Specifically, an initial load of one job per processor is queued to a small subset ($\log P$) of processors during the first cycle of execution. Therefore, a light system load exists throughout the experiment. The entire simulation requires 40.0 secs (4.0 secs for each execution cycle), if load balancing is effective. Note that 0.2 sec is required to process the initial load plus $12.96 \times 9 \times 0.2$ secs to process the remaining cycles. This totals 23.528 secs, leaving an average of 1.647 secs of idle time per cycle.

5.2 Performance Metrics

The data and bar charts included in Figs. 9–11 measure the comparative performance of the various load-balancing algorithms on a 32-processor SGI Origin2000. The X-axis of the bar charts show the number of processors used. The Y-axis tracks the following metrics:

Message Traffic Comparison: The total number of balancing and distribution messages that were exchanged during the simulation.

Total Jobs Transferred: The total number of jobs that were transferred from one processor to another. If a job is transferred multiple times before execution, each transfer is individually counted. Note that it may have been appropriate to count multiple job transfers only once since an actual data transfer would incur most of the overhead. However, the Total Jobs Transferred metric as defined can be useful in that it gives an indication of the flexibility of an algorithm: its ability to adapt to a rapidly changing dynamic load environment. For example, load-balancing algorithms that do not allow multiple transfers would be the least flexible and thus expected to generate the smallest values for Total Jobs Transferred. Thus, a load-balancing scheme returning a high value of this metric is not necessarily undesirable.

Maximum Variance in Idle Time: The difference in processing time (in secs) between the busiest processor and the least busy processor.

Total Time to Complete: The total amount of elapsed time (in secs) before all jobs are fully processed.

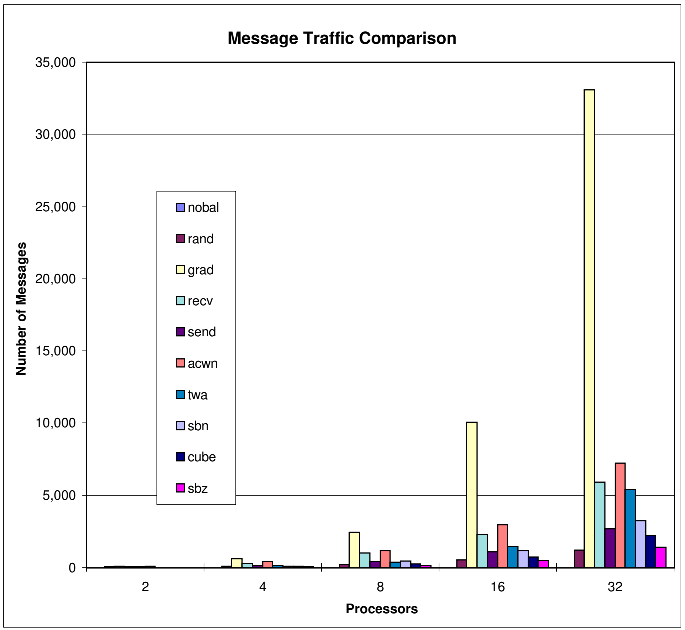
5.3 Summary of Results

As mentioned earlier, the Basic SBN-based load-balancing algorithm (in short, SBN), its Heuristic Variant (SBZ), and the Tree Walking algorithm (TWA) were implemented using the SBN topology, while the other load-balancing schemes were implemented assuming a hypercube topology. Recall that the Hypercube Variant of SBN (CUBE) utilizes the Basic SBN algorithm adapted for the hypercube. Analyzing CUBE, we can determine whether performance improvements are due to the proposed load-balancing algorithms or due to the SBN topology. The following paragraphs analyze each of the four performance metrics measured in our experiments.

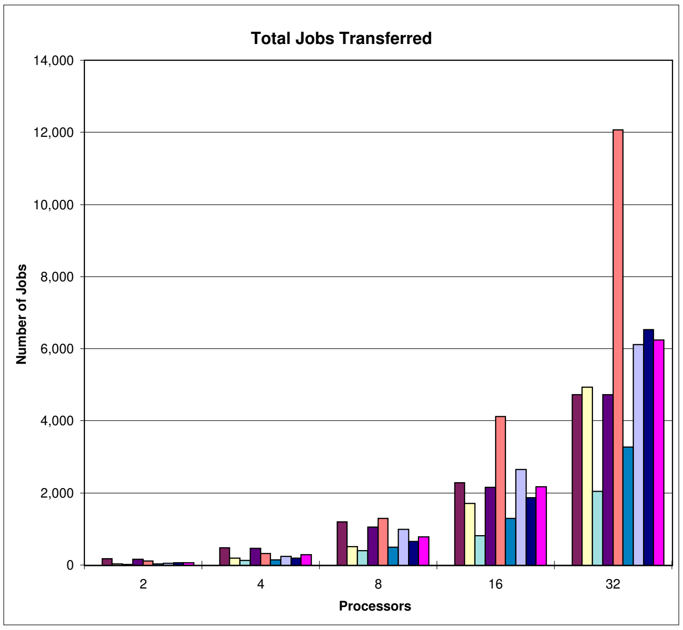
With respect to the Message Traffic Comparison metric, the Gradient algorithm (GRAD) generates, by far, the largest amount of message traffic. The Receiver Initiated algorithm (RECV) also generates a large number of messages because of its tendency to become unstable under light system loads. Idle processors can flood the system with job request messages in situations where their neighbors do not have excess jobs to transfer. To alleviate this condition, we introduced a 0.1 sec delay between job requests. Longer delays tend to reduce the load-balancing effectiveness of RECV under light loads. As expected, SBZ generates less message traffic than the other two SBN schemes. Likewise, all SBN-based algorithms incur less communication than TWA. In general, the algorithms that do the worst in terms of load balancing require little or no message communication. For example, no load balancing (NOBAL) does not generate any message traffic, proving that some interaction among processors is necessary to balance a system load.

Next consider the Total Jobs Transferred metric. During the heavy system load test (cf. Fig. 9), the Total Jobs Transferred values are significantly less than that in the other experiments (cf. Figs. 10 and 11). This is due to the fact that during heavy loads, most processors are busy and not seeking additional jobs to process. Under the light system load test (cf. Fig. 11), the situation changes. Here, the SBN-based algorithms generate the largest values because of their tendency to pass jobs through more processors in order to satisfy those with low loads. It is important to realize that the data associated with jobs need to be transferred only once, just before a job is about

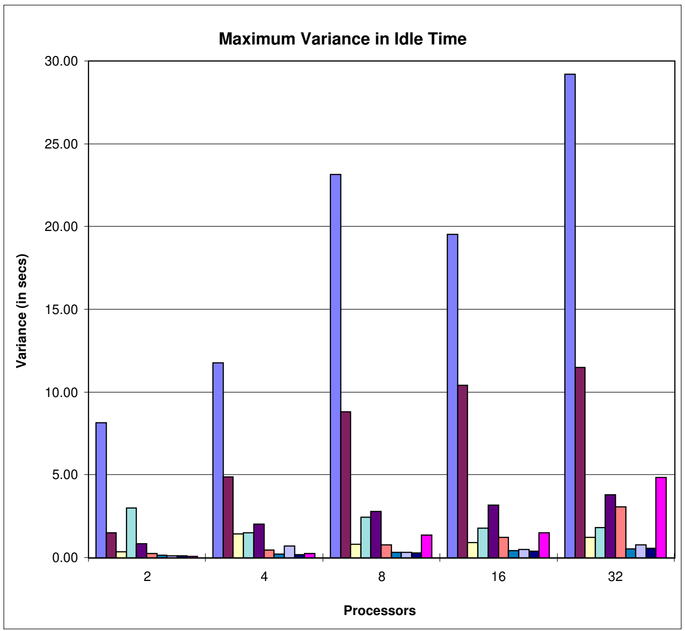
P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	0	20	63	25	34	99	18	12	14	8
4	0	71	586	287	130	381	102	85	75	43
8	0	207	2,426	1,005	398	1,164	364	457	227	138
16	0	515	10,050	2,279	1,074	2,936	1,443	1,152	703	460
32	0	1,202	33,065	5,923	2,688	7,228	5,390	3,230	2,183	1,390
average	0	403	9,238	1,904	865	2,362	1,463	987	640	408



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	0	183	33	12	155	109	31	42	60	65
4	0	472	195	124	459	325	142	237	189	294
8	0	1,195	505	395	1,049	1,298	495	983	656	790
16	0	2,282	1,709	818	2,157	4,122	1,298	2,649	1,874	2,169
32	0	4,731	4,939	2,041	4,720	12,068	3,272	6,118	6,527	6,246
average	0	1,773	1,476	678	1,708	3,584	1,048	2,006	1,861	1,913



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	8.13	1.51	0.36	3.00	0.84	0.25	0.14	0.10	0.10	0.08
4	11.76	4.87	1.42	1.49	2.03	0.44	0.20	0.70	0.19	0.25
8	23.15	8.79	0.81	2.44	2.80	0.77	0.30	0.33	0.29	1.37
16	19.53	10.41	0.90	1.77	3.16	1.21	0.41	0.47	0.40	1.48
32	29.19	11.47	1.23	1.82	3.78	3.06	0.52	0.75	0.54	4.83
average	18.35	7.41	0.94	2.10	2.52	1.15	0.31	0.47	0.30	1.60



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	23.39	20.42	16.09	21.22	19.24	19.37	19.67	18.44	17.93	20.04
4	24.61	20.58	18.03	19.94	21.11	19.38	19.38	20.00	18.57	20.81
8	30.03	25.68	18.68	19.17	20.96	20.40	19.36	19.16	18.04	18.96
16	30.95	23.92	18.76	18.95	20.41	19.74	19.21	18.64	19.40	20.16
32	34.42	24.10	20.11	19.78	21.26	21.19	19.60	18.33	19.80	19.93
average	28.68	22.94	18.33	19.81	20.60	20.02	19.44	18.91	18.75	19.98

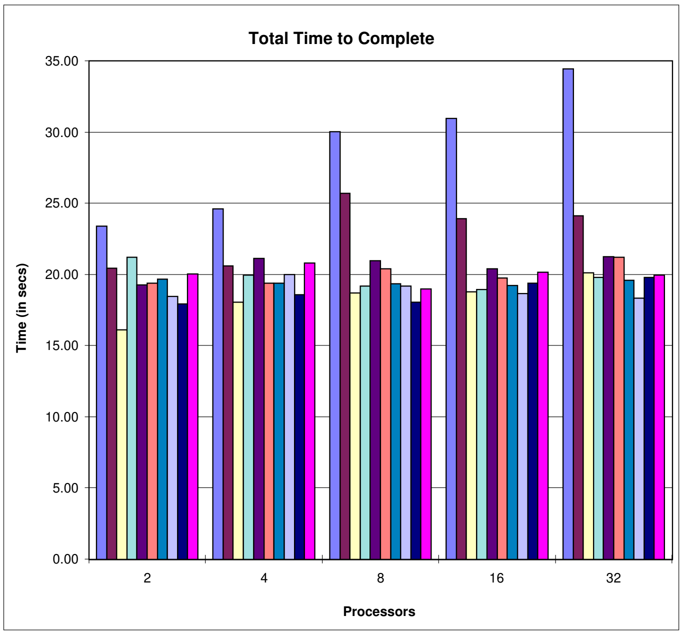
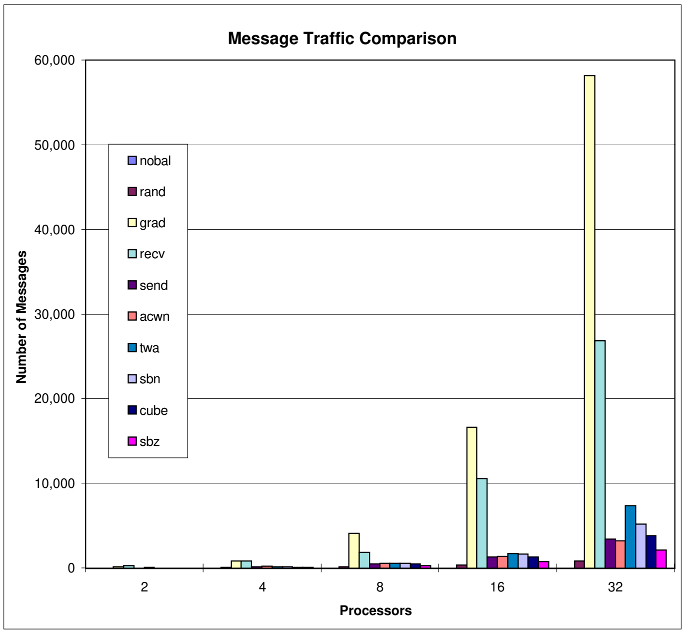
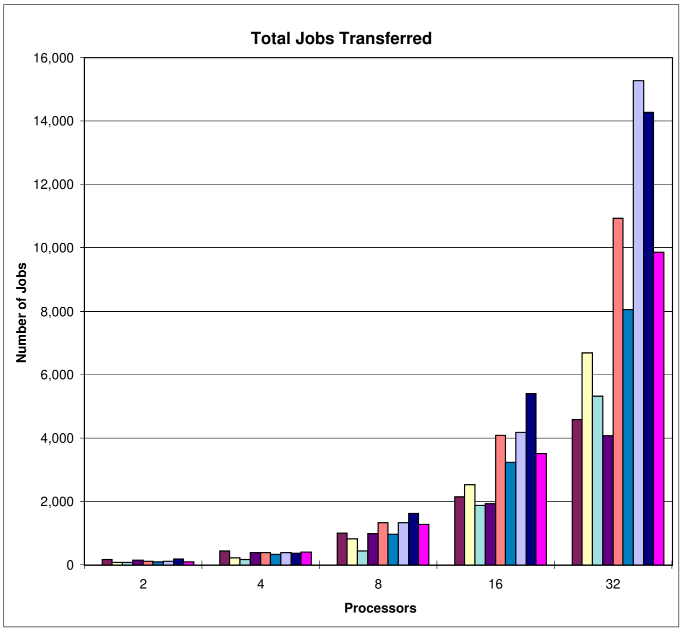


Figure 9: Performance under heavy system load.

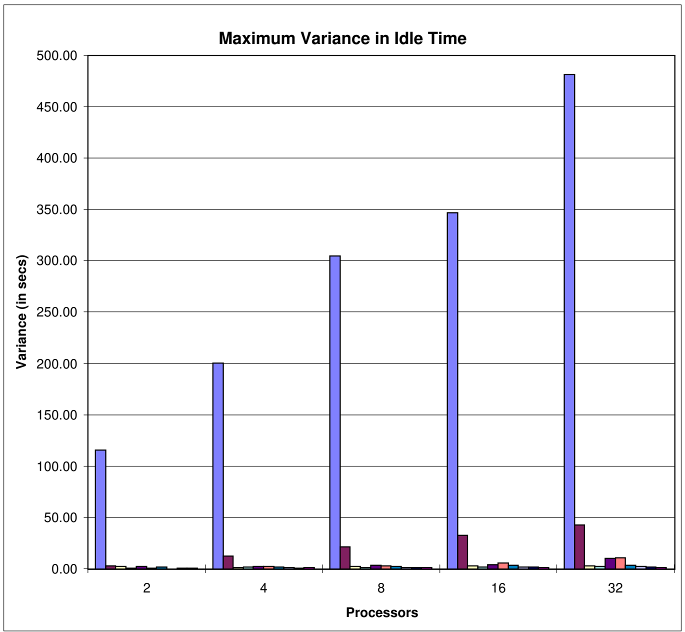
P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	0	14	139	274	33	48	21	22	31	16
4	0	45	815	848	163	178	121	137	90	95
8	0	147	4,095	1,863	447	534	538	519	495	247
16	0	351	16,601	10,537	1,318	1,367	1,730	1,648	1,326	729
32	0	834	58,155	26,821	3,428	3,216	7,375	5,185	3,824	2,128
average	0	278	15,961	8,069	1,078	1,069	1,957	1,502	1,153	643



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	0	167	77	76	146	100	87	110	180	99
4	0	438	219	166	373	385	333	385	372	392
8	0	1,004	810	442	976	1,329	956	1,334	1,625	1,272
16	0	2,136	2,520	1,863	1,930	4,085	3,231	4,179	5,399	3,500
32	0	4,574	6,683	5,320	4,077	10,933	8,043	15,271	14,278	9,863
average	0	1,664	2,062	1,573	1,500	3,366	2,530	4,256	4,371	3,025



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	115.55	2.96	2.21	0.30	2.29	0.58	1.47	0.25	0.38	0.48
4	200.23	12.51	1.19	1.65	2.47	2.44	1.51	1.16	0.71	0.89
8	304.77	21.41	2.06	1.34	3.51	2.65	2.41	1.21	1.07	1.03
16	347.08	32.62	2.84	1.82	3.87	5.41	3.24	1.63	1.42	1.10
32	481.26	42.38	2.68	2.26	10.11	10.46	3.21	2.26	1.41	1.12
average	289.78	22.38	2.20	1.47	4.45	4.31	2.37	1.30	1.00	0.92



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	276.28	41.02	41.21	40.86	42.15	40.72	43.38	43.20	40.02	39.90
4	329.78	42.15	40.45	41.39	40.01	39.95	39.97	40.35	40.94	39.90
8	371.73	46.20	40.17	41.11	40.72	40.71	39.97	40.22	40.19	40.00
16	393.45	52.00	39.91	39.96	40.09	40.04	39.94	40.08	39.98	39.92
32	494.20	60.41	39.99	40.09	42.14	40.75	40.09	39.93	39.98	39.91
average	373.09	48.36	40.35	40.68	41.02	40.43	40.67	40.76	40.22	39.93

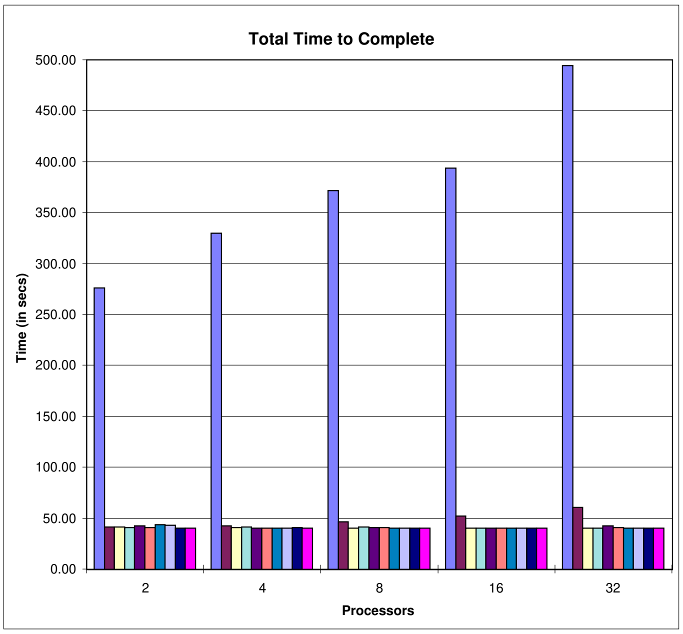
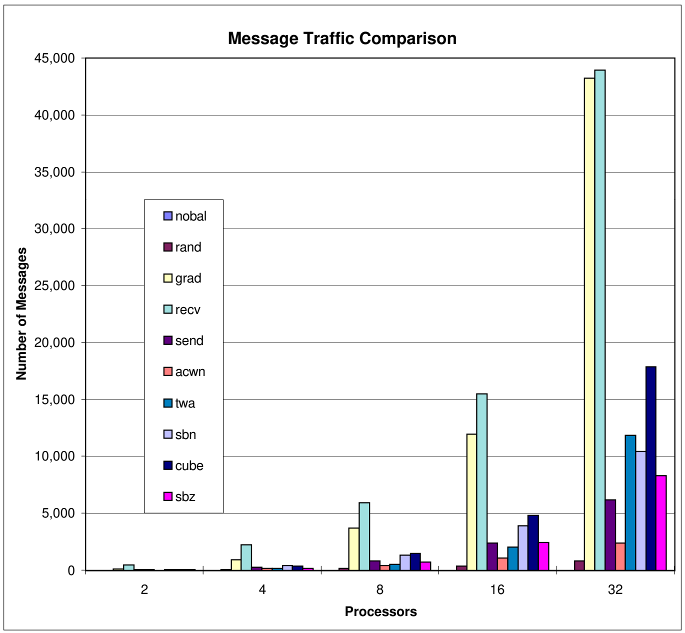
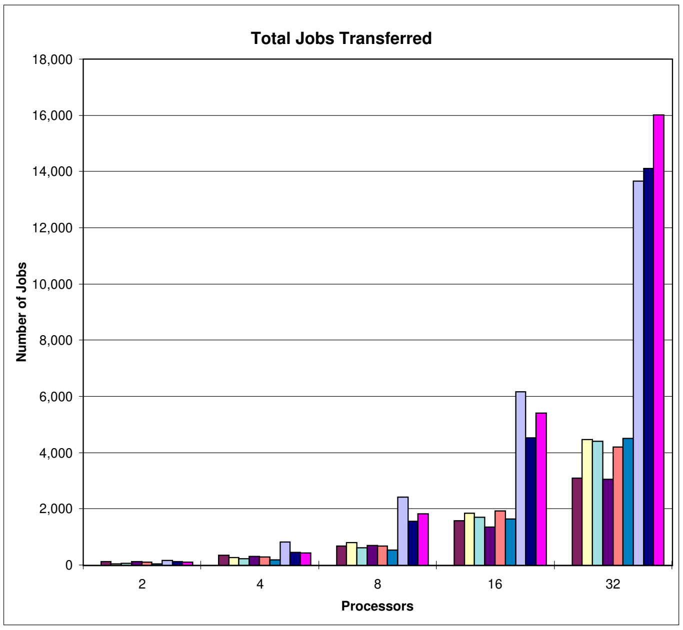


Figure 10: Performance when transitioning from heavy to light system load.

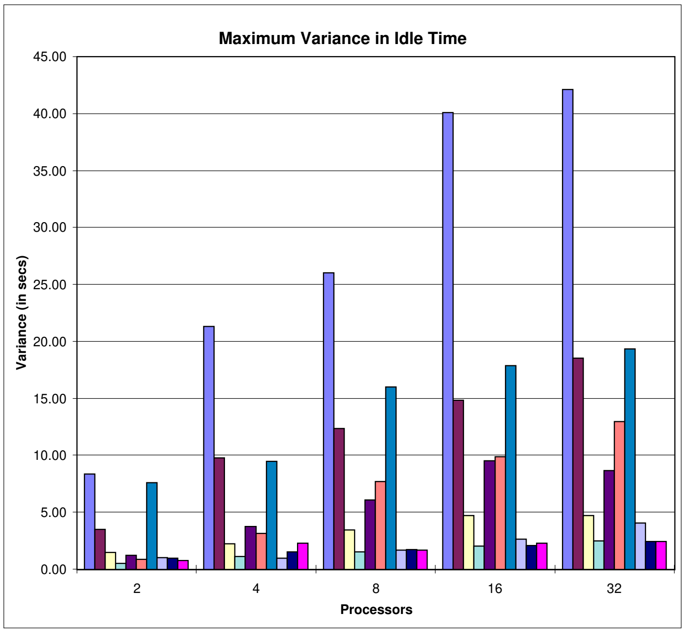
P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	0	14	87	466	46	42	24	47	60	27
4	0	47	898	2,235	235	149	127	401	371	148
8	0	138	3,718	5,930	805	404	500	1,336	1,462	712
16	0	337	11,921	15,501	2,395	1,045	2,026	3,917	4,810	2,433
32	0	805	43,243	43,950	6,151	2,400	11,862	10,414	17,881	8,313
average	0	268	11,973	13,616	1,926	808	2,908	3,223	4,917	2,327



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	0	127	47	54	120	101	47	160	129	98
4	0	338	262	218	304	291	194	827	455	422
8	0	676	796	624	689	672	535	2,420	1,549	1,821
16	0	1,580	1,851	1,693	1,346	1,929	1,645	6,172	4,523	5,405
32	0	3,093	4,454	4,411	3,044	4,197	4,509	13,668	14,118	16,007
average	0	1,163	1,482	1,400	1,101	1,438	1,386	4,649	4,155	4,751



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	8.35	3.51	1.49	0.52	1.21	0.85	7.60	1.02	0.98	0.78
4	21.29	9.78	2.21	1.13	3.77	3.13	9.48	0.94	1.50	2.30
8	26.02	12.33	3.42	1.52	6.07	7.67	15.98	1.69	1.74	1.65
16	40.10	14.85	4.71	2.02	9.50	9.89	17.89	2.61	2.09	2.28
32	42.11	18.52	4.73	2.47	8.67	12.94	19.33	4.06	2.44	2.44
average	27.57	11.80	3.31	1.53	5.84	6.90	14.06	2.06	1.75	1.89



P	nobal	rand	grad	recv	send	acwn	twa	sbn	cube	sbz
2	40.94	39.90	40.11	39.92	40.69	41.01	40.55	39.91	39.96	39.92
4	42.04	40.82	39.91	39.91	40.01	39.93	39.95	39.97	39.92	39.96
8	42.67	39.96	39.93	39.94	40.01	39.92	42.48	39.89	39.97	39.93
16	47.73	39.98	39.97	39.96	40.01	39.95	40.70	39.90	39.96	39.97
32	49.05	39.95	39.96	39.95	40.02	39.91	41.54	39.95	39.97	39.98
average	44.49	40.12	39.98	39.94	40.15	40.14	41.04	39.92	39.96	39.95

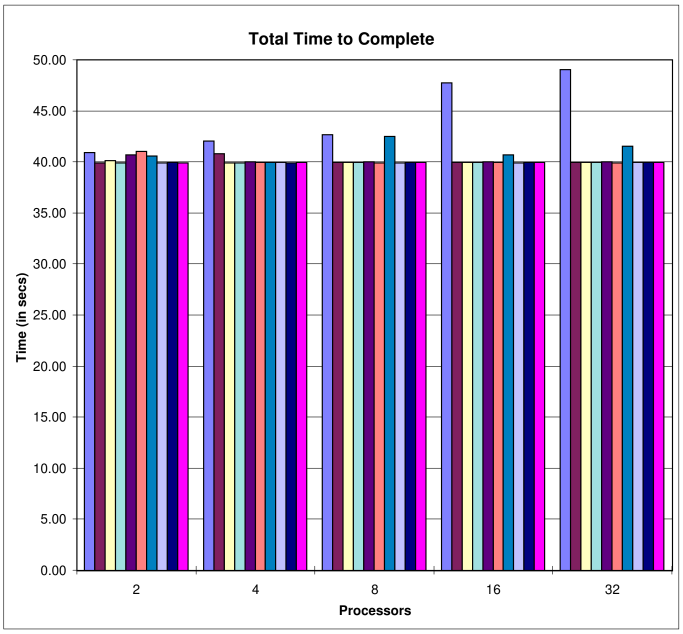


Figure 11: Performance under light system load.

to execute. Note also that the SBN algorithms utilize bulk transfers in sending distribution messages to relocate jobs. These characteristics reduce the negative effects of message latency and minimize the additional overhead that is incurred. As would be expected, the Random (RAND) and Sender Initiated (SEND) algorithms that allow only one job transfer before execution, have the smallest Total Jobs Transferred values. However, an interesting result is that Adaptive Contracting (ACWN), which also does not allow jobs to be rerouted more than once, has a higher Total Jobs Transferred measure. This is due to the fact that ACWN does more work to balance the load initially (when jobs arrive) than the other algorithms. In fact, during the heavy system load test (cf. Fig. 9), Total Jobs Transferred values for ACWN is among the highest.

When considering the Maximum Variance in Idle Time metric, NOBAL obviously performs the worst by far. RAND, although reducing the idle time, is much less effective than the others. SEND and ACWN have similar performance, which is somewhat better than RAND. Note that RAND, ACWN, and SEND do not allow multiple job migrations. This feature prevents these algorithms from efficiently adapting to a dynamic job execution environment. TWA shows a large imbalance under light system loads (cf. Fig. 11) that could stem from its lack of using a minimum processor workload threshold.

The last metric is the Total Time to Complete. Recall that the amount of work is increased proportionately with the number of processors (e.g., twice as much processing is required for $P = 8$ than for $P = 4$). Under light loads (cf. Fig. 11), only NOBAL and TWA fail to complete within the optimal amount of time (40.0 secs). This is consistent with the results discussed above where both approaches show a large variance in processor idle time. Similarly, under the heavy-to-light load test (cf. Fig. 10), most of the algorithms finish at near-optimal times (approximately 40.0 secs). Here, only NOBAL and RAND could not process the job queues within the expected time. If we observe the values from the chart in Fig. 9 for the Total Time to Complete, the best average results for the heavy system load test were recorded by SBN, CUBE, and GRAD.

To compare the overall performance of the SBN-based algorithms to the other approaches, first note that the performance of CUBE is very similar to that of SBN. This indicates that our comparisons are fair even though the topology of SBN is different from that of the other algorithms. The similarity in performance between CUBE and SBN is not surprising since both topologies have a depth of $\log P$ and use the same basic balancing approach.

To continue with our performance comparison, consider the heavy system load (cf. Fig. 9) experiment. In this test, Total Time to Complete is the most important metric. Clearly, NOBAL and RAND are the worst performers, and are hence non-competitive. Looking at the average performance of the experiment, we find that SEND, ACWN, and SBZ are worse than the average performance of SBN and CUBE by 6% to 10%.

We next look at the experimental results under the transition from heavy to light system load (cf. Fig. 10) and the light system load (cf. Fig. 11) scenarios for the remaining five algorithms

(GRAD, RECV, TWA, SBN, and CUBE). In these two tests, Message Traffic Comparison is the most important metric since we expect the completion times to be near-optimal for all reasonable load-balancing approaches. Figures 10 and 11 show that GRAD and RECV have significantly greater message traffic than TWA and the SBN-based algorithms.

While comparing TWA to the SBN algorithms, we also want to determine how much load-balancing overhead is incurred by TWA. This is important because TWA suspends application processing during balancing. Our results show that TWA spends between 3.23% and 4.86% of the execution time in balancing a network of 32 processors. With a network of 16 processors, the overhead varies between 0.73% and 1.08%. Based on the superlinear increase in the fraction of time spent by TWA in balancing the load, the overhead could potentially become intolerable for large networks of processors. By contrast, SBN hides all of the load-balancing time since processing is never suspended.

Based on the above analysis, we conclude that the SBN approach is a viable alternative and compares favorably with other load-balancing algorithms. All three SBN strategies are effective because global load information is obtained from all processors to balance the system load. Most of the other load-balancing algorithms work only locally with processors interacting with their immediate neighbors; therefore, the load information is likely to be less accurate. Even though the SBN approach is global, the load-balancing overhead is significantly reduced because the depth of an SBN of P processors is $\log P$. Furthermore, the fact that SBN-based algorithms allow application processing to continue uninterrupted during load balancing makes them latency tolerant: most of the communication and data distribution overhead is hidden under processing.

6 Conclusions

In this paper, we have proposed three new load-balancing algorithms (Basic, Hypercube Variant, and Heuristic Variant) based on a topology-independent logical communication pattern among processors, called a symmetric broadcast network (SBN). A detailed experimental investigation with synthetic workloads showed that this approach to load balancing compares favorably with several other schemes. The metrics that measure the Maximum Variance in Idle Time and Total Time to Complete, demonstrate that all three algorithms are effective in balancing the system load while optimizing the completion and idle times. The Message Traffic Comparison metric shows that use of the Heuristic Variant reduces the overhead associated with load balancing traffic when compared to the two other SBN-based algorithms.

The Basic SBN algorithm has been extended and effectively applied to a dynamic adaptive-mesh application to balance processor workloads while significantly reducing the data redistribution costs [4]. This optimization was possible by overlapping processing and workload migration. The latency-tolerance feature makes the SBN approach a natural choice for grid and cluster computing environments consisting of heterogeneous computers. The SBN topology also provides fault tolerance that would allow applications to continue correct execution while using resources that are constantly changing. These will be the focus of future research.

Acknowledgements

This work was supported in part by Texas Advanced Research Program Grant Number TARP-97-003594-013 and by NASA Ames Research Center under Cooperative Agreement Number NCC 2-5395. The authors would like to thank Prof. Michael Palis (JPDC SAE) and the anonymous referees for their constructive and detailed reviews of the paper.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to Algorithms," MIT Press, Cambridge, MA, 1989.

[2] G. Cybenko, Dynamic load balancing for distributed-memory multiprocessors, *J. Parallel Distrib. Comput.* **7** (1989), 279–301.

[3] S. K. Das, D. J. Harvey, and R. Biswas, Adaptive load-balancing algorithms using symmetric broadcast networks: Performance study on an SP2, *in* "Proc. 26th International Conference on Parallel Processing, Bloomingdale, IL, 1997," pp. 360–367.

[4] S. K. Das, D. J. Harvey, and R. Biswas, Parallel processing of adaptive meshes with load balancing, *IEEE Trans. Parallel Distrib. Systems* **12** (2001), to appear.

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

[6] 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.

[7] D. L. Eager, E. D. Lazowska, and J. Zahorjan, Adaptive load sharing in homogeneous distributed systems, *IEEE Trans. Software Engineering* **12** (1986), 662–675.

[8] D. L. Eager, E. D. Lazowska, and J. Zahorjan, A comparison of receiver-initiated and sender-initiated adaptive load sharing, *Performance Evaluation* **6** (1986), 53–68.

[9] R. Elsässer, A. Frommer, B. Monien, and R. Preis, Optimal and alternating-direction loadbalancing schemes, *in* "EuroPar'99 Parallel Processing (P. Amestoy *et al.*, Eds.), Lecture Notes in Computer Science, Vol. 1685, pp. 280–290, Springer-Verlag, Berlin, 1999."

[10] M. R. Eskicioglu, Design issues of process migration facilities in distributed systems, *in* "Scheduling and Load Balancing in Parallel and Distributed Systems, IEEE Computer Society Press, Los Alamitos, CA, 1995," pp. 414–424.

[11] G. Fox, A. Kalawa, and R. Williams, The implementation of a dynamic load balancer, *in* "Proc. Conference on Hypercube Multiprocessors, 1987," pp. 114–121.

[12] Y. F. Hu and R. J. Blake, An improved diffusion algorithm for dynamic load balancing, *Parallel Comput.* **25** (1999), 417–444.

[13] P. Krueger and M. Livny, The diverse objectives of distributed scheduling policies, *in* "Proc. 7th International Conference on Distributed Computing Systems, Berlin, Germany, 1987," pp. 242–249.

[14] F. C. H. Lin and R. M. Keller, The gradient model load balancing method, *IEEE Trans. Software Engineering* **13** (1987), 32–38.

[15] R. Lüling, B. Monien, and F. Ramme, Load balancing in large networks: A comparative study, *in* "Proc. 3rd Symposium on Parallel and Distributed Processing, Dallas, TX, 1991," pp. 686–689.

[16] S. Pulidas, D. F. Towsley, and J. A. Stankovic, Embedding gradient estimators in load balancing algorithms, *in* "Proc. 8th International Conference on Distributed Computing Systems, San Jose, CA, 1988," pp. 482–490.

[17] K. G. Shin and Y. C. Chang, Load sharing in hypercube multicomputers for real-time applications, *in* "Proc. Conference on Hypercubes, Concurrent Computers, and Applications, 1989," pp. 617–621.

[18] 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.

[19] N. G. Shivaratri, P. Krueger, and M. Singhal, Load distributing for locally distributed systems, *IEEE Computer*, **25** (1992), 33–44.

[20] W. Shu and M.-Y. Wu, Runtime incremental parallel scheduling (RIPS) on distributed memory computers, *IEEE Trans. Parallel Distrib. Systems* **7** (1996), 637–649.

[21] J. Song, A partially asynchronous and iterative algorithm for distributed load balancing, *Parallel Comput.* **20** (1994), 853–868.

[22] C.-Z. Xu and F. C. M. Lau, Analysis of the generalized dimension exchange method for dynamic load balancing, *J. Parallel Distrib. Comput.* **16** (1992), 385–393.

[23] C.-Z. Xu and F. C. M. Lau, “Load Balancing in Parallel Computers: Theory and Practice,” Kluwer, Boston, MA, 1997.

[24] C.-Z. Xu, F. C. M. Lau, B. Monien, and R. Lüling, Nearest-neighbor algorithms for load balancing in parallel computers, *Concurrency: Practice and Experience* **7** (1995), 707–736.