Design of Novel Load-Balancing Algorithms with Implementations on an IBM SP2

Sajal K. Das[1], Daniel J. Harvey[1], and Rupak Biswas[2]

[1] CS Dept., University of North Texas, Denton, TX 76203, USA
[2] NASA Ames Research Center, Moffett Field, CA 94035, USA

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

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 processor 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 [8] relates to decisions made at compile time, and compile-time programming tools are necessary to adequately estimate the required resources. Dynamic algorithms [1, 5, 6, 7] allocate/reallocate resources at run time based on one or more system parameters. Determining which parameters to maintain and how to broadcast them are important design considerations.

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

Recently, Das et. al. [2, 3, 4] have suggested a different approach to dynamic load balancing, by introducing a logical topology-independent communication pattern called symmetric broadcast network (SBN). We refine this approach and propose three novel and efficient load-balancing algorithms, one of which is adapted for use on the hypercube architecture. Based on their operational characteristics, our SBN-based algorithms can be classified (e.g. [9]) as:

Adaptive: performance is adapted to the average number of queued jobs;
Symmetrically initiated: senders and receivers can initiate load balancing;
Stable: the network is not burdened with excessive load-balancing traffic;
Effective: system performance is not degraded when the algorithms operate.

The three proposed algorithms are implemented on an IBM SP2, using the Message-Passing Interface (MPI). Performance is analyzed based on simulation using Poisson-distributed synthetic loads under various metrics: total number of jobs transferred, total completion time, message traffic per node, and maximum variance in node idle time. Empirical results of our extensive experiments demonstrate that the load balancing achieved by the SBN approach is superior to several other existing techniques such as Random [5], Gradient [7], Receiver Initiated [6], Sender Initiated [6], and Adaptive Contracting [6].

2 General Characteristics of SBNs

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

- A single node forms the basis network SBN(0).
- For $d > 0$, SBN(d) is obtained from a pair of SBN($d-1$)s by adding a communication stage in the front and extra interprocessor links as follows: (a) node i in stage 0, is connected to node j=(i+$P/2$) mod 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.

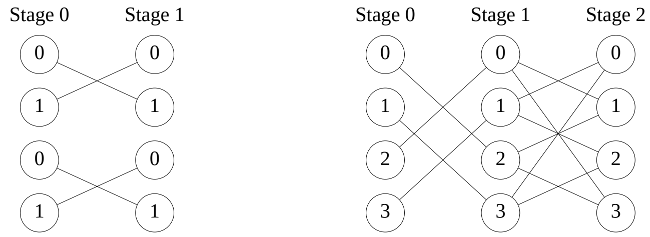

Fig. 1. Construction of an SBN(2) from a pair of SBN(1)s

The SBN approach defines unique communication patterns among the nodes in the network. For any node at stage 0 as the source, there are $\log P$ stages of communication with each node appearing exactly once. The successors and predecessors for each node are uniquely defined by specifying the originating node and the communication stage.

As an example, consider the communication pattern for SBN(3) shown in Fig. 2 that is used for messages originating from node 0. In general, if n_x^s is the corresponding node in the communication pattern for messages originating from node x, then $n_x^s = n_0^s \oplus x$, where $\oplus$ is the exclusive-OR operator. Thus, all SBN communication patterns can be derived from the template corresponding to the one with node 0 as the root. The predecessor and successors to node n_0^s are:

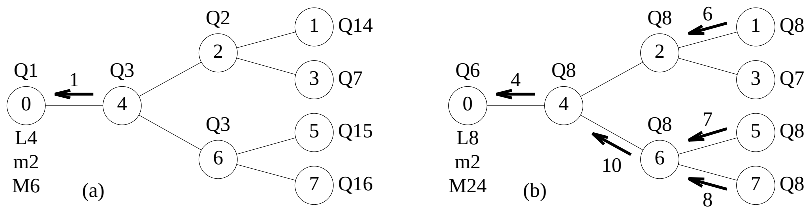

Fig. 2. An example of load balancing using the standard SBN algorithm

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 a possible SBN communication pattern, but many others can be easily derived based on network topology and application requirements. In [4], the SBN approach was adapted for use on the hypercube using a modified binomial spanning tree to ensure that all successor and predecessor nodes at any communication stage are adjacent nodes in the hypercube.

All SBN 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`, the minimum and maximum system load levels. The system load level `SysLL` is the average number of jobs queued per processor. If a processor has a queue length `QLen` below `MinTh`, a message is initiated to begin load balancing. If `QLen` is larger than `MaxTh`, extra jobs are distributed through the network. If this distribution overloads other processors, load balancing is triggered. Algorithm behavior is affected by the values chosen for `MinTh` and `MaxTh`. For instance, `MinTh` must be large enough so that sufficient jobs can be received before a lightly-loaded processor becomes idle; however, it should not be too big so 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 in the network, very little load-balancing activity should be required.

Two types of messages are processed in the SBN approach. The first type are balancing messages that indicate unbalanced system load. These messages originate from an unbalanced node and are 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 type of messages is for job distribution and used for two purposes. First, they are used to route the current `SysLL` through the network. Each node, upon receipt of such a message, updates its estimate of the average number of jobs queued per node in the network. The system thresholds, `MinTh` and `MaxTh`, are also updated. 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`. It can also be in response to a predecessor's need for jobs. This need is embedded in the load-balance messages as well as in the distribution messages that respond to these operations. To reduce message traffic, a node does not initiate additional load-balancing activity until all previous messages that have passed through the node have been completely processed.

3 SBN-Based Load-Balancing Algorithms

3.1 Standard SBN Algorithm

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

To illustrate the processing involved in a load-balancing operation, consider the SBN(3) in Fig. 2(a). The id and `QLen` for each node are shown. For example, node 6 has three jobs queued, indicated as Q3. The initial values of `SysLL`, `MinTh`, and `MaxTh` at node 0 are 4, 2, and 6, respectively (indicated as L4, m2, and M6). After a load-balancing request is sent through the SBN and then routed back to node 0, these values are updated to 8, 2, and 24, respectively, using:

$$\begin{aligned}
\mathtt{SysLL} &= \lceil \mathtt{TotalJobsQueued} \,/\, \mathtt{P} \rceil, \\
\mathtt{MinTh} &= \min\,(\mathtt{MinTh},\ \mathtt{SysLL}{-}1), \\
\mathtt{MaxTh} &= \mathtt{SysLL} + 2^{\lfloor \mathtt{SysLL} \,/\, \mathtt{MinTh} \rfloor}.
\end{aligned}$$

Note that when load balancing is initiated, node 4 distributes half of its `QLen`, i.e., $\lfloor 3/2 \rfloor$ job, back to node 0. This is shown by the arrow in Fig. 2(a).

Distribution messages are then used to route excess jobs to the successor nodes or to indicate a need for jobs if the local `QLen` is less than `SysLL`. Jobs are routed back to the predecessor nodes when appropriate. Figure 2(b) shows the result of this distribution. The arrows indicate jobs routed between nodes. To load balance P processors, a total of $3P{-}3$ messages have to be processed.

3.2 Hypercube Variant

The SBN approach can be adapted for implementation on a hypercube topology, using the modified binomial spanning tree. A complete description of this hypercube variant is given in [4]. It operates in a manner similar to the standard SBN algorithm with the following differences:

- `SysLL` is computed when all balance messages arrive at the final node in the network. This is possible because there is a unique final 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.
- 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.

3.3 Heuristic SBN Algorithm

Both previous algorithms process a large number of messages to accurately maintain `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 SBN algorithm, a processor estimates `SysLL` by averaging `QLen` for the processors through which the balance message has passed. An appropriate number of jobs are then returned to the predecessor nodes as follows:

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

If `ExJobs` $= 0$ or if `SysLL` > 2 when `ExJobs` $= 1$, the balance message is forwarded to the next stage. Otherwise, the load balancing is terminated.

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

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

4 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 jobs during 10 job creation cycles. The number of jobs generated at each node during each cycle follows a Poisson distribution. By changing the parameter λ, both heavy and light system load conditions are dynamically simulated. Jobs are processed by "spinning" for the designated time period. The simulation terminates when all jobs have been processed. Three test runs are reported here:

Heavy System Load (cf. Fig. 3): Initially, 10 jobs per node are randomly distributed throughout the network. Jobs generated during execution are more than that the network can process. Job duration averages one second.

Transition from Heavy to Light System Load (cf. Fig. 4): Fifty jobs multiplied by the number of processors are distributed to a small subset of nodes as an initial load. A light load of jobs is generated as the load-balancing algorithms proceed. Job duration averages two seconds. Note that the initial load imbalance also needs to be corrected.

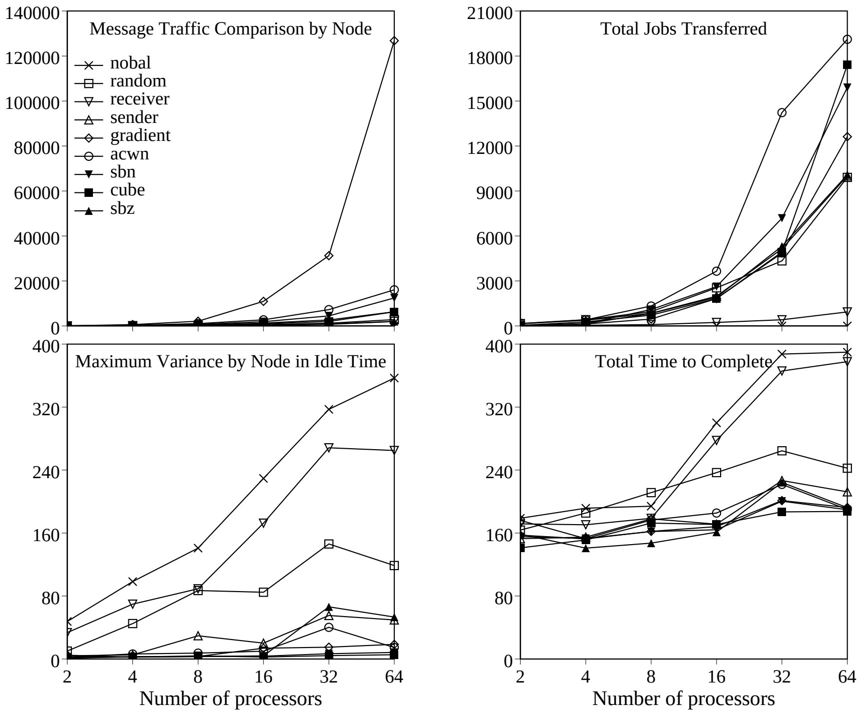

Fig. 3. Heavy system load

Light System Load (cf. Fig. 5): A small number of jobs are initially distributed to a small subset of nodes. A light load of jobs are created as the load-balancing algorithms operate.

The data and line charts in Figs. 3–5 measure the performance of the various load-balancing algorithms on an SP2, using the following variables:

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

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

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

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 (*random*) algorithm, although significantly reducing the idle time, is less effective than the remaining algorithms. The Sender Initiated (*sender*) algorithm balances the load more evenly than *random*; however, the Receiver Initiated (*receiver*) algorithm does better only when the system load is light. For light to moderate loads, *receiver* 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

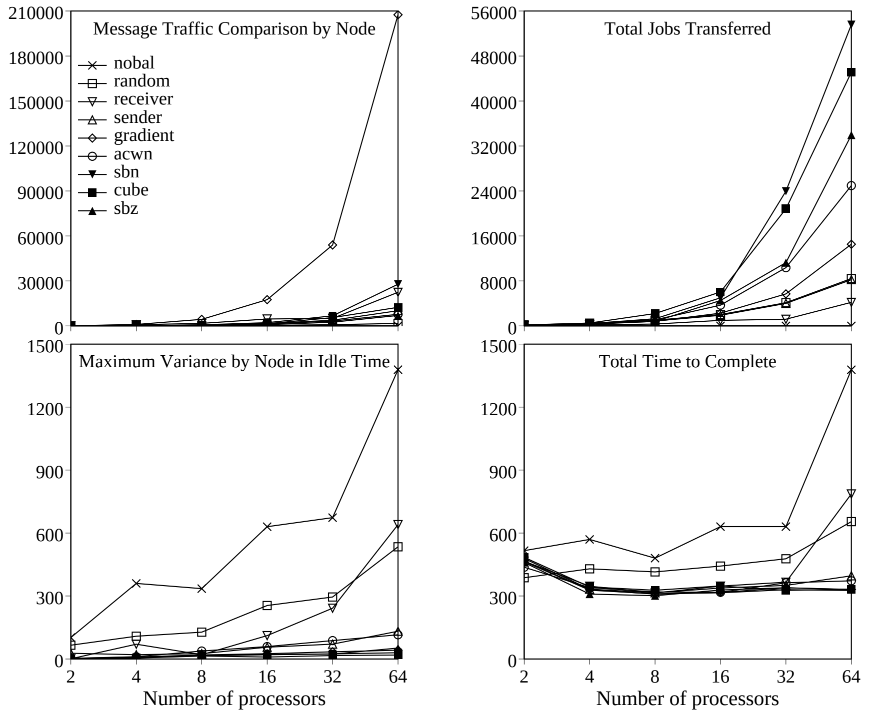

Fig. 4. Transition from heavy to light system load

the cost of increasing the idle time. At heavy system loads, *sender* 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 overloaded nodes. Also significant communication is required to update neighbor node information, often resulting in excessive network traffic. The Adaptive Contracting (*acwn*) algorithm performs the best in periods of heavy system loads. However, as is true for *gradient*, the system traffic and the number of jobs migrated increase.

Both the standard SBN (*sbn*) algorithm and its hypercube variant (*cube*) are able to balance the system load more evenly than others. Their performance characteristics are very similar. Both require less message traffic than *gradient* but cause a higher number of job migrations, especially in 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 system load as well as *cube* or *sbn*. This is because its estimate of `SysLL` is not necessarily accurate. For light loads, *sbz* transfers more jobs than the other algorithms; however, it requires fewer messages than *gradient*, *sbn*, or *cube*. Overall, the empirical results demonstrate that the SBN-based approach to dynamic load balancing is an effective one.

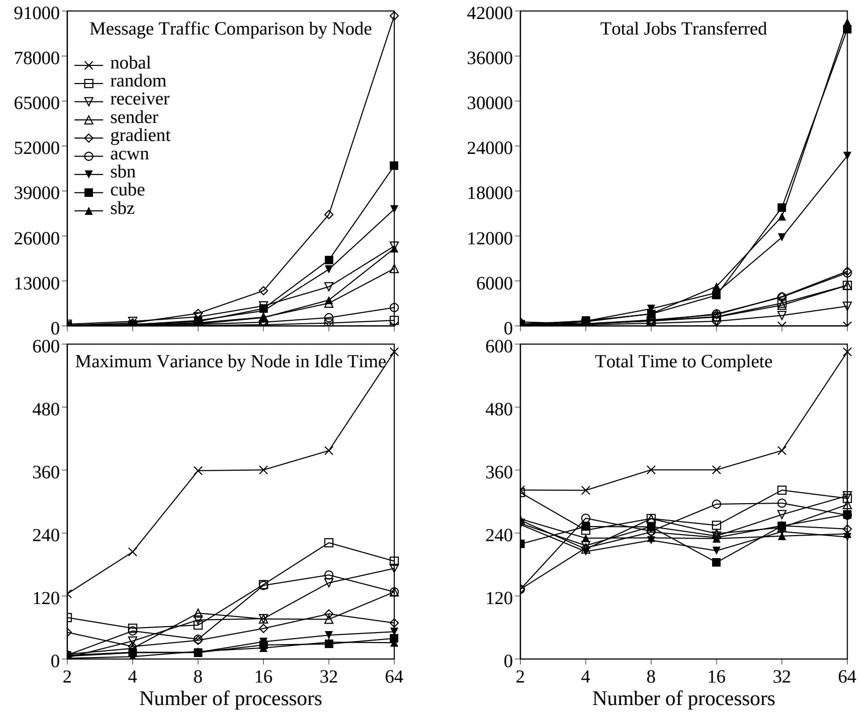

Fig. 5. Light system load

References

1. Cybenko, G.: Dynamic load balancing for distributed-memory multiprocessors. J. Parallel Distrib. Comput. **7** (1989) 279–301
2. Das, S., Prasad, S.: Implementing task ready queues in a multiprocessing environment. International Conference on Parallel Computing (1990) 132–140
3. Das, S., Yang, C., Leung, N.: Implementation of load balancing in multiprocessor systems using a symmetric broadcast network. International Conference of Parallel and Distributed Systems (1992) 589–596
4. Das, S., Harvey, D., Biswas, R.: Adaptive load-balancing algorithms using symmetric broadcast networks, NASA Ames Research Center Technical Report NAS-97-014 (1997)
5. Eager, D., Lazowska, G., Zahorjan, J.: Adaptive load sharing in homogeneous distributed systems. IEEE Trans. on Soft. Engrg. **12** (1986) 662–675
6. Eager, D., et al.: A comparison of receiver initiated and sender initiated adaptive load sharing. Perf. Eval. **6** (1986) 63–68
7. Lin, F., Keller, R.: The gradient model load balancing method. IEEE Trans. on Soft. Engrg. **13** (1987) 32–38
8. Sarkar, V., Hennessy, J.: Compile-time partitioning and scheduling of parallel programs. Scheduling and Load Balancing in Parallel and Distributed Systems (1995) 61–70
9. Shivaratri, N., Krueger, P., Singhal, M.: Load distributing for locally distributed systems. Computer **25** (1992) 33–44