How Network Topology Affects Dynamic Load Balancing

Peter Kok Keong Loh, Wen Jing Hsu, Cai Wentong, and Nadarajah Sriskanthan
Nanyang Technological University

The authors compare the performances of five dynamic load-balancing strategies. The simulator they've developed lets them measure these performances across a range of network topologies, including a 2D mesh, a 4D hypercube, a linear array, and a composite Fibonacci cube.

A multiprocessor network without load balancing processes processor-generated tasks locally with little or no sharing of computational resources. Load balancing, on the other hand, uses a multiprocessor network's inherently redundant processing power by redistributing the workload among the processors to improve the application's overall performance.

Load-balancing strategies fall broadly into either *static* or *dynamic* classifications. A network with static load balancing computes task information, such as execution time (execution cost), from the application before load distribution. The network distributes tasks once, before execution, and the allocation stays the same throughout the application's execution. A network with dynamic load balancing uses little or no a priori task information, and must satisfy changing requirements by making task-distribution decisions during runtime. For certain applications, dynamic load balancing is preferable, because then the problem's variable behavior more closely matches available computational resources. But dynamic load balancing incurs communication overheads that are topology-dependent (where *topology* is the interconnection structure of the multiprocessor network).

Researchers have proposed several load-balancing strategies.[1–9] However, in most cases, these researchers made performance comparisons using either a simulated distributed computer system[5,8,9] or a multiprocessor network with a specific topology.[3,7,8] We have developed a topology-independent simulator to compare the performances of five well-known, dynamic load-balancing strategies: the Gradient Model

1063-6552/96/$4.00 © 1996 IEEE

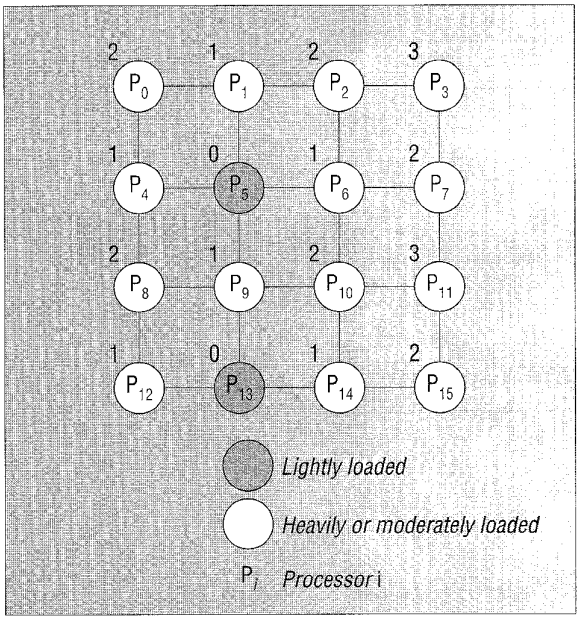


Figure 1. Proximity distribution.

(GM) strategy,[1] the Sender-Initiated (SI) and Receiver-Initiated (RI) strategies,[2,7] the Central Job Dispatcher (LBC)[4] strategy, and the Prediction-Based (Pred)[5,9] strategy. In this article, we compare their performances across a series of 16-node networks of different topologies: a 4×4 mesh, a 4D hypercube, a linear array, and a composite Fibonacci cube.[10]

The Gradient Model strategy

In this strategy, every processor interacts only with its immediate neighbors. Basically, lightly loaded processors inform other processors in the system of their state, and overloaded processors respond by sending a portion of their load to the nearest lightly loaded processor in the system.

When execution begins, every processor computes its total load. Two threshold values gauge whether a processor is lightly, heavily, or moderately loaded. A processor with a total load below the *low water mark* is considered lightly loaded. One that exceeds the *high water mark* is heavily loaded, and one where the total load is in-between is moderately loaded.

In this strategy, *proximity* defines the minimum distance between the current processor and the nearest lightly loaded processor in the network (see Figure 1). We measure interprocessor distances (and, thus, proximity values) in terms of the number of hops, where a *hop* is the distance between any two directly connected processors. We will assume that all hops are the same length. The figure gives the proximity for each processor.

Every processor in the network initially sets its proximity to d_{max}, a constant equal to the network's diameter, or the largest distance between two processors in the network. A processor's proximity is set to zero if it becomes lightly loaded. All other processors P_i with nearest neighbors n_j compute their proximity as

$$\text{proximity}(P_i) = \min(\text{proximity}(n_j)) + 1$$

A processor's proximity cannot exceed d_{max}. A system is saturated and does not require load balancing if all processors report a proximity of d_{max}. If a processor's proximity changes, that processor must notify its immediate neighbors. Hence, lightly loaded processors, reporting a proximity of zero, initiate the load-balancing process. The gradient map of the proximities of all processors in the system routes tasks between overloaded and underloaded processors.

The Sender-Initiated strategy

Here, an overloaded processor (sender) trying to send a task to an underloaded processor (receiver) initiates load distribution. Derek Eager, Edward Lazowska, and John Zahorjan proposed three fully distributed sender-initiated strategies.[2] The difference in these strategies is the policy used in locating the processors to transfer or receive tasks. In the first strategy, the network simply transfers a task to a randomly selected processor without any information exchange between the processors aiding the decision. The second strategy is similar but with the introduction of a threshold value to prevent tasks from being transferred to an overloaded processor. In the third strategy, the network polls a number of randomly selected processors and compares their load sizes. The network then transfers the task to the processor with the smallest load.

These strategies, however, have several major disadvantages. They have no mechanism to ensure that the lightly loaded processor selected is a moderate distance away from the heavily loaded processor. Task transfers between two distant processors can result in performance degradation during load balancing. Furthermore, the lightly loaded processor selected on the basis of load size might not necessarily be the best candidate, because the polling mechanism arbitrarily polls randomly selected processors. To ensure consistency in performance comparison with the GM and the RI strategies, we have adopted a sender-initiated strategy, proposed by Marc Willebeek-LeMair and Anthony Reeves,[7] which also uses only immediate neighbor state information.

This sender-initiated strategy uses a nearest-neighbor approach with overlapping neighborhood domains to achieve global load balancing over the network. A preset threshold L_{high} identifies the sender. An overloaded processor performs load balancing whenever its load level l_i is greater than the threshold value—that is, when $l_i > L_{high}$. Once the sender is identified using the threshold, the next step is to determine the amount of load (number of tasks) to transfer to the sender's neighbors. The average load L_{avg} in the domain is

$$L_{avg} = \frac{1}{K+1}\left(l_p + \sum_{k=1}^{K} l_k\right)$$

where l_p is the load of the overloaded sender, K is the total number of immediate neighbors, and l_k is the load of Processor k. The network assigns each neighbor a weight h_k, according to

$$h_k = \begin{cases} L_{avg} - l_k & \text{if } l_k < L_{avg} \\ 0 & \text{otherwise} \end{cases}$$

These weights are summed to determine the total deficiency H_d:

$$H_d = \sum_{k=1}^{K} h_k$$

Finally, we define the proportion of Processor p's excess load, which is assigned to neighbor k as δ_k, such that

$$\delta_k = \lceil (l_p - L_{avg}) h_k / H_d \rceil$$

where $\lceil x \rceil$ stands for the largest integer value of x. Once the network has determined the quantity of load to migrate, it dispatches the appropriate number of tasks.

Figure 2 shows an example of the SI strategy, where surplus load is transferred to its underloaded neighbors. Here, we assume that the threshold L_{high} is taken as 10. Hence, the network identifies Processor A as the sender and does its first calculation of the domain's average load:

$$L_{avg} = \frac{0+5+20+7+8}{5} = 8$$

The weight for each neighborhood processor is then as follows:

Processor	B	C	D	E
Weight, h_k	8	3	1	0

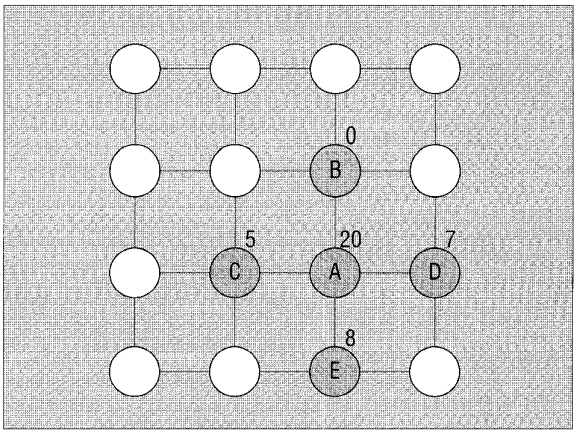


Figure 2. Example of SI strategy in a 4 × 4 mesh.

Summing these weights determines the total deficiency:

$$H_d = 8 + 3 + 1 + 0 = 12$$

The proportions of Processor A's load that are assigned to its neighbors are

Processor	B	C	D	E
Load, δ_k	8	3	1	0

The final load on each processor is therefore 8.

The Receiver-Initiated strategy

The RI strategy is like the converse of the SI strategy in that the receiver, rather than the sender, initiates load balancing. Moreover, the threshold value is lower in the RI strategy. The underloaded processors in the network handle the load-balancing overhead, which can be significant in a heavily loaded network.

In this strategy, the network identifies, as the receiver, a processor whose load size falls below the threshold value L_{low}. The receiver handles task migration by requesting proportional amounts of load from immediate overloaded neighbors. The network assigns each neighbor k a weight h_k, according to the following formula:

$$h_k = \begin{cases} l_k - L_{avg} & \text{if } l_k > L_{avg} \\ = 0 & \text{otherwise} \end{cases}$$

We sum these weights to determine the total surplus H_s. Processor p then determines a load portion δ_k to be migrated from its neighbor k:

$$\delta_k = (L_{avg} - l_p) h_k / H_s$$

Finally, Processor p sends respective load requests to its specific neighbors.

Figure 3 shows an example of the RI strategy, where the network transfers surplus load from a processor's

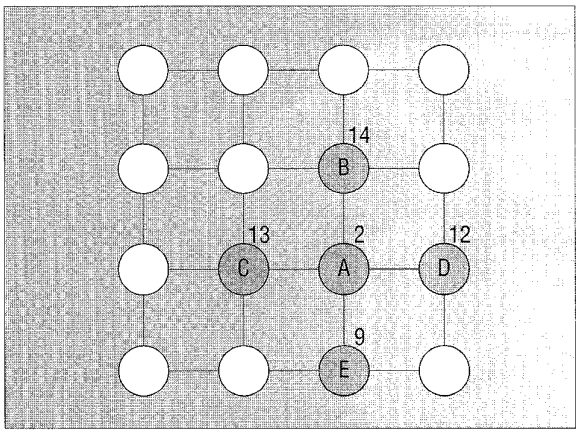


Figure 3. Example of RI strategy in a 4 × 4 mesh.

overloaded neighbors. We assume here that the L_{low} threshold is 6 and that Processor A is the receiver. The network does its first calculation of the average domain load:

$$L_{avg} = \frac{14+13+2+12+9}{5} = 10$$

The weight for each neighborhood processor is then as follows:

Processor	B	C	D	E
Weight, h_k	4	3	2	0

We sum these weights to determine the total surplus:

$$H_s = 4 + 3 + 2 + 0 = 9$$

The proportion of load that Processor A requests from each neighboring processor is

Processor	B	C	D	E
Load, δ_k	4	3	2	0

We tabulate the final load on each processor as follows:

Processor	A	B	C	D	E
Load	11	10	10	10	9

The dynamic load-balancing strategies discussed so far use local (neighboring domain) state information to guide load distribution. The processor selection and task-transfer policies are distributed in nature: all processors in the network have the responsibility of achieving global load balance. However, these strategies do not try to locate the best *transfer partner* (destination processor).

A strategy that uses global (network wide) state information can usually identify the most suitable transfer partner. We now present one such strategy.[4]

The Central Task Dispatcher strategy

In this strategy, one of the network processors acts as a centralized job dispatcher. The dispatcher keeps a table containing the number of waiting tasks in each processor. Whenever a task arrives at or departs from a processor, the processor notifies the central dispatcher of its new load state.

When a state-change message is received or a task-transfer decision is made, the central dispatcher updates the table accordingly. The network bases load balancing on this table and notifies the most heavily loaded processor to transfer tasks to a requesting processor. The network also notifies the requesting processor of the decision. With this strategy, there could be greater communication overheads with larger networks, because the decision making is no longer distributed.

In the original strategy, a processor would send a task request when it started its operation with no local job or when it became idle. However, in designing the simulation environment, we introduced a threshold value L_{low}, which is equivalent to the lower water mark of the GM strategy. This accounts for scenarios where some processors start off with an average load. In this case, when a processor's load goes below L_{low}, the network embeds the state-change message with a task request tag, so that the message serves the dual purpose of table update and load request at the central task dispatcher.

The Prediction-Based strategy

In recent years, some researchers have focused their efforts on prediction-based, dynamic load-balancing strategies.[5,9] These strategies stem from predicted process requirements for achieving load balancing.

The prediction-based strategy proposed by Kumar Goswami, Murphy Devarakonda, and Ravishankar Iyer has demonstrated prediction of the CPU, memory, and I/O requirements of a process, before its execution, using a statistical pattern-recognition method.[5] However, even though the predicted values are close to the actual ones, this strategy incurs significant computation overheads. Moreover, the prediction mechanism uses network-dependent task identifier numbers to tabulate the possible resource requirements.

Other researchers have proposed a strategy that uses task-transfer probabilities to predict a processor's load requirements.[9] Probability models are more realistic,

because they can capture a distributed scheduling application's time-varying characteristics. Another advantage is that the network can estimate a processor's load at any time without querying that processor. We have adopted this strategy in our simulation.

This prediction-based strategy uses service time $S_i(t)$ as the load index to perform dynamic load balancing. Each processor estimates its own service time for the next time interval and broadcasts this information to all other processors. During a given time interval Δt, the network can estimate the service time $S_i(t)$ by recording the total time used by the processor i in servicing tasks, and the number of task departures completed during that interval.

Therefore, at a specific time t, we have

$$S_i(t) = \Delta t / d_i(t)$$

where $S_i(t)$ is the service time per task, and $d_i(t)$ is the total number of task departures in Δt. Each processor distributes this information to all other processors and computes the mean service time $S_m(t)$:

$$S_m(t) = \frac{S_1(t) + S_2(t) + \mathrm{K} + S_n(t)}{n}$$

where n is the total number of processors in the network, and $S_m(t)$ is the mean service time for the network. Each processor then determines the load status of itself and other processors using $S_m(t)$, as follows:

$$S_i(t) > S_m(t) \Rightarrow \text{heavily loaded}$$
$$S_i(t) < S_m(t) \Rightarrow \text{lightly loaded}$$

The next step involves determining $W(t)$, the ratio of excess service time to the mean service time, on each heavily loaded processor i:

$$W_i(t) = \frac{S_i(t) - S_m(t)}{S_i(t)}$$

Finally, each processor i computes and maintains a list of task-transfer probabilities between itself and all other underloaded processors j in the network:

$$p_{ij} = \left(\frac{\sum_{k=1}^{L} S_k(t) - S_j(t)}{\sum_{k=1}^{L} S_k(t)} \right) W_i(t)$$

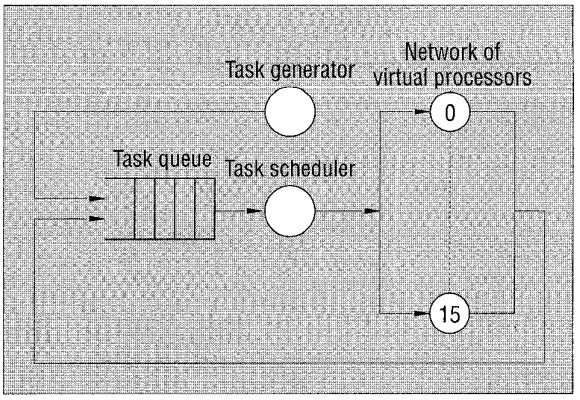


Figure 4. The simulator's system configuration.

where L is the total number of lightly loaded processors. The heavily loaded processor selects the lightly loaded processor with the highest transfer probability. The number of tasks to be transferred is proportional to $W_i(t)$.

Simulation model

We have developed a simulator based on a study by Songnian Zhou,[8] which other researchers have further verified and examined.[5,11] We employ a trace-driven simulation approach. In this approach, job traces are collected from a production distributed computer system and used to simulate a loosely coupled multiprocessor network. The distributed system, consisting of a Unix-based VAX-11/780 host, supports both research and academic applications of staff and students.

To ensure that the measurements applied to homogeneous processors, we restricted the trace-collection efforts to one host. Figure 4 shows the simulator's configuration.

The task scheduler implements the corresponding dynamic load-balancing strategy. It also randomly distributes tasks in the network of virtual processors initially and handles the runtime migration of tasks. The task scheduler inserts tasks to be migrated back into the task queue for rescheduling in a different virtual processor.

Dynamic load balancing involves two basic types of overhead costs. First, the network must measure the processors' current load levels, and exchange messages so that other processors recognize them. Second, the network must make placement decisions and transfer tasks between the processors. The simulator's design includes the following parameters: task size, computation cost, communications cost, and task migration cost. These vary according to the computing environment or platform. On the basis of experimental measurements, therefore, we set at 10 milliseconds the cost for computing various values such as threshold levels or current load levels of CPU time. We assigned a cost of 10 ms to the transferring node, and the receiving node took

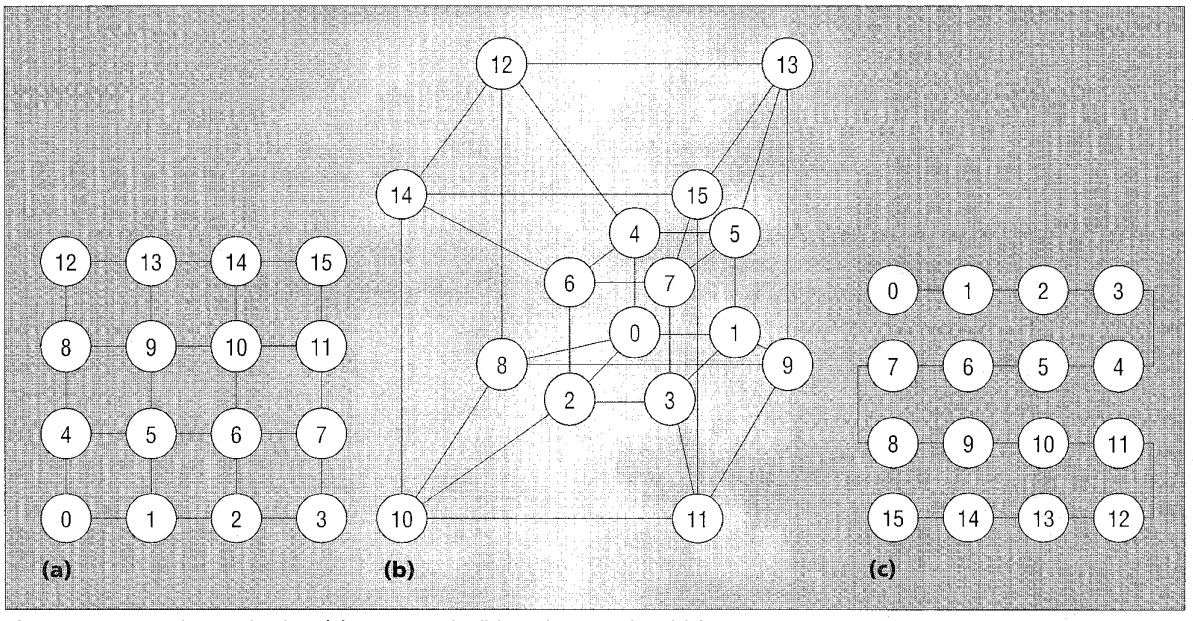


Figure 5. Network topologies: (a) 4 × 4 mesh; (b) 4D hypercube; (c) linear array.

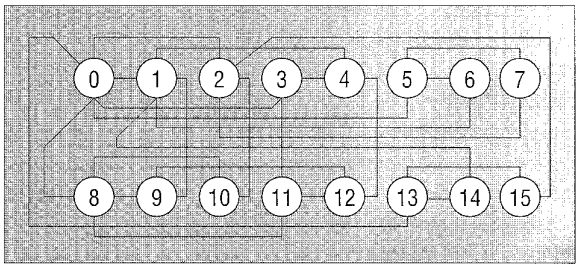


Figure 6. A composite Fibonacci cube.

10 ms to process the information. We assigned 100 ms of CPU time for a task transfer for both the sending and receiving processors, causing a 200-ms execution delay to the task being transferred.

Zhou's study has shown that 60 to 65% of the tasks have execution times below 500 ms. In most cases, only about 25% of network processors have loads at least 10% higher than average. Hence, the tasks used for simulation have execution times ranging from 200 to 800 ms. The computer system randomly generates each task's execution time. Each simulation run uses 1,600 tasks (about 100 tasks per processor node), and two initial task-distribution approaches are adopted. The first approach simulates a stable situation, where the network randomly assigns about 100 tasks to each processor. The eventual outcome is that no idle processor is present in the network, and about 25 to 35% of the total processors are overloaded.

The second task-distribution approach creates a highly unstable network system, where some processors are heavily loaded and others can have a task size of zero. These scenarios let us examine the algorithmic reliability of the load-balancing strategy and the variation in topological parameters.

Performance metrics

In general, performance is an absolute measure described in terms of response time, utilization, or any other objective function specified. In our research, performance analysis represents *normalized performance* and *stabilization time*.

Normalized performance Π determines the effectiveness of the load-balancing strategy (such that $\Pi \rightarrow 0$ if the strategy is ineffective and $\Pi \rightarrow 1$ if the strategy is effective). This is a comprehensive metric; it accounts for the initial level of load imbalance as well as the load-balancing overheads. We formally define Π as

$$\Pi = \frac{T_{\text{nolb}} - T_{\text{bal}}}{T_{\text{nolb}} - T_{\text{opt}}}$$

where T_{nolb} is the time to complete the work on a multiprocessor network without load balancing, T_{opt} is the time to complete the work on one processor divided by the number of processors in the network, and T_{bal} is the time to complete the work on the multiprocessor network with load balancing. When the load-balancing time approaches the optimal time ($T_{\text{bal}} \rightarrow T_{\text{opt}}$), then $\Pi \rightarrow 1$. On the other hand, if load balancing is poor and does not improve the network much over the case without load balancing, then $T_{\text{bal}} \rightarrow T_{\text{nolb}}$ and $\Pi \rightarrow 0$.

Stabilization time or load-balancing time indicates how long the network takes to achieve a balanced state where no further task transfers are required. A low stabilization time doesn't necessarily indicate an efficient or compre-

Table 1. Network topological parameters.

Topology	Δ_{AVG}	Number of nodes with degree 1	2	3	4	5	6	Θ_{AVG}
4 × 4 mesh	2.67	-	4	8	4	-	-	3.00
4D hypercube	2.13	-	-	-	16	-	-	4.00
Fibonacci cube	2.41	-	5	7	2	1	1	3.13
Linear array	5.67	2	14	-	-	-	-	1.88

hensive strategy. It could also indicate that, because of inadequate information, the load-balanced network is suboptimal. Such a network can still have an unevenly distributed workload even though the imbalance is insufficient to trigger load-redistribution activities.

Our objective here is not to select the best algorithm but to compare the variations in performance of each strategy over different network topologies. In particular, we are interested in the effects of topological parameters, such as interprocessor distances and connectivity, on load-balancing performance with varying load levels.

Network topologies

We compare the performances of the five dynamic load-balancing strategies on a 4 × 4 mesh, a 4D hypercube, a linear array, and the Fibonacci cube. The Fibonacci cube is both a subgraph of the hypercube and a supergraph of several common topologies (see "Background on Fibonacci cube" sidebar). It serves as an interesting comparison with the other three more common topologies, which are illustrated in Figure 5.

The other network topologies in this research have 16 nodes. A Fibonacci cube, however, supports only f_n nodes—that is, a Fibonacci number of nodes. Hence, the linking of node pairs with unity Hamming distance combines the Fibonacci cubes Γ_6, Γ_5, and Γ_4 to form a composite topology of 16 nodes, as Figure 6 shows.

Topological parameters

A *hop* is the distance between any two directly connected processors in the network. The distance between two processor nodes i and j in a network G of size N is the number of hops in the shortest path connecting i and j. The network's *diameter* is the largest distance, in terms of the number of hops, between two processors. Evaluating the network diameter, however, does not give a global picture of the network, because a small number of hops can separate many of the network's other processors even when the diameter is large. An example of this is the 4 × 4 mesh topology, which has a diameter of 6. The average interprocessor distance, on the other hand, illustrates the global topological view of the network. We define the *average processor distance* Δ_{avg} as

$$\Delta_{avg} = \sum_{i \neq j}^{G} \frac{D(i,j)}{N(N-1)}$$

where, for a 4 × 4 mesh, $N = 16$.

The *node degree* is the number of links incident on a processor node. By the same reasoning, we define the *average node degree*, denoted by Θ_{avg}, as the sum of the node degrees divided by the number of network processors. Table 1 lists the values of these topological parameters for the four topologies we are considering.

Intuitively, a network topology with a smaller average processor distance has lower communication overheads between processor pairs. A network that has a higher average node degree has more directly connected neighbors per processor.

Simulation results (normalized performance)

Figure 7a (stable network) and Figure 7b (unstable network) illustrate the simulation results for normalized performance versus topology. The legend for each graph shows the representative symbols for the respective dynamic load-balancing strategy.

The normalized performances of the RI, the SI, and the GM strategies in a stable network are better than the LBC and the Pred strategies for mesh, hypercube, and Fibonacci topologies (see Figure 7a). The first three strategies use local domain (immediate connected neighbors) state information and employ distributed processor selection and task-transfer policies. The LBC and the Pred, however, use global domain (network) state information and centralized processor selection and task transfer. In the stable situation with no idle processors and a fractional overloading (25 to 35%), the network localizes load-balancing activities to arbitrary regions, favoring distributed policies that use local domain information.

In the linear array, however, the communication overheads and the reduction in local-domain computational resources incurred because of the structure's linearity take their toll on these strategies, causing the performances of the RI, the SI, and the GM to fall below those of the LBC and the Pred. The LBC strategy, using a centralized dispatcher, is less significantly affected by topological parameter variations for networks of

similar size. In addition, the higher accuracy of the transfer processor identification in the LBC outweighs the overheads incurred. Notwithstanding, there is also performance degradation.

The Pred strategy, like the GM, requires periodic state updates on processors and uses distributed processor selection and task-transfer policies. On average, however, the processor selection of the Pred is more accurate than with the GM, enabling the Pred to partially overcome the topological constraints and perform slightly better.

In an unstable network (see Figure 7b), the extent of load balancing increases. The ranking changes, with the RI strategy still maintaining the lead but now being followed by the LBC and the Pred. Here, a processor-selection policy of higher accuracy produces better load balancing. The accuracy of selection heavily depends on the domain information's comprehensiveness, a dependency that favors the global domain schemes of the LBC and the Pred. Nevertheless, the greater communication and computation overheads caused by the frequent state information broadcasts and updates lets the RI strategy maintain its lead.

The results show that in all strategies, regardless of the network situation, network topologies with lower Δ_{avg} and higher Θ_{avg} yield better performances. A lower Δ_{avg} minimizes communications overhead and therefore task migration costs. A higher Θ_{avg} means more computational resources in the local domain are available, favoring the dissemination and exchange of local domain information.

Simulation results (stabilization time)

Figure 8a (stable network) and Figure 8b (unstable network) illustrate simulation results for stabilization time versus topology.

Background on Fibonacci cube

The Fibonacci cube is a new interconnection topology, first proposed by Wen Jing Hsu[1], which derives from the well-known Fibonacci series. The nth Fibonacci number is as follows:

$f_0 = 0$
$f_1 = 1$
$f_n = f_{n-1} + f_{n-2}$ where $n \geq 2$

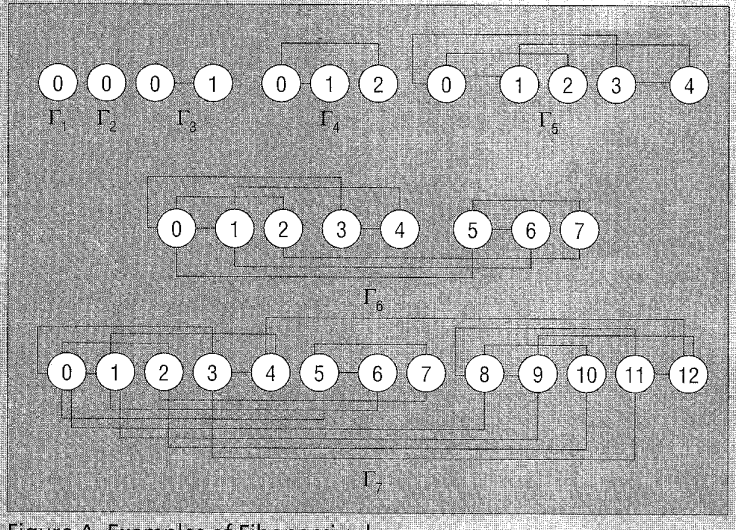


Figure A. Examples of Fibonacci cubes.

Zeckendorf's theorem states that any natural number can be uniquely represented as a sum of Fibonacci numbers. Formally, we assume i is an integer, where $0 \leq i \leq f_n - 1$ and $n \geq 3$. Let i be represented by an order-n Fibonacci code $(b_{n-1}, \ldots, b_3, b_2)_F$, where b_j is either 0 or 1 for $2 \leq j \leq (n-1)$ and

$$i = \sum_{j=2}^{n-1} (b_j f_j)$$

We obtain the Fibonacci representation of i by first finding the greatest Fibonacci number $f_k \leq N$. Then we assign a 1 to the bit corresponding to f_k. We continue the procedure recursively for $N - f_k$. The unassigned bits are 0s. In the Fibonacci code, the rightmost bit corresponds to f_2 (see Table A).

A Fibonacci cube of order n, denoted by Γ_n, consists of f_n nodes. We can consider Γ_n a graph (comprising vertices V and edges E) consisting recursively of two subgraphs Γ_{n-1} and Γ_{n-2}. Formally, Γ_n is a graph $(V(f_n), E(f_n))$, where $V(f_n) = \{0, 1, \ldots, f_n - 1\}$. If i and j are two vertices in this graph, then $(i, j) \in E(f_n)$ if and only if $H(I_F, J_F) = 1$. I_F and J_F are the Fibonacci code representations of i and j, and $H(I_F, J_F)$ is the Hamming distance between I_F and J_F.

Figure A illustrates some examples of Fibonacci cubes. Each node in Γ_{n-2} is con-

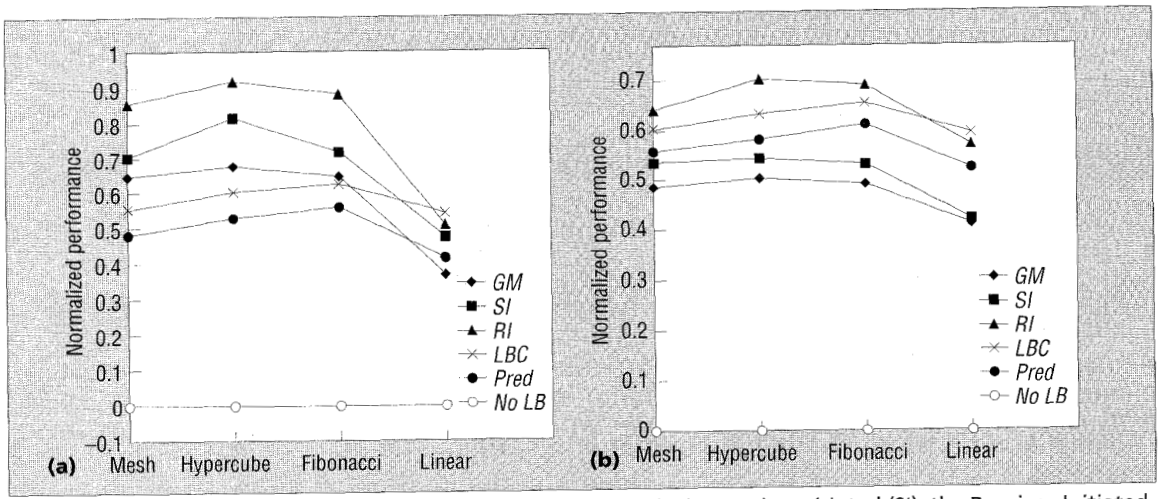


Figure 7. Normalized performances for the Gradient Model (GM), the Sender-Initiated (SI), the Receiver-Initiated (RI), the Central Job Dispatcher (LBC), and the Prediction-Based (Pred) strategies, and for no load balancing (No LB): (a) a stable network; (b) an unstable network.

For a given topology, the stabilization time required by a load-balancing strategy depends on the loading of processors responsible for the task transfer. For the stable situation (Figure 8a), the RI and the LBC strategies, where lightly loaded processors invoke task transfers, have a longer stabilization time. This is because these strategies employ lower threshold values than do the SI, the GM, or the Pred strategy. The network invokes the

Table A. Fibonacci code representations.

Decimal number	Fibonacci number	Fibonacci code
0	0	000 000
1	1	000 001
2	1	000 010
3	2	000 100
4	3	000 101
5	5	001 000
6	8	001 001
7	13	001 010
8	21	010 000
9	34	010 001
10	55	010 010
11	89	010 100
12	144	010 101
13	233	100 000
14	377	100 001
15	610	100 010

nected to a counterpart node in Γ_{n-1}. Γ_0 is an empty graph, and Γ_1 is a graph with one node.

Each graph in Figure A represents a multiprocessor interconnection network, with the labeled node representing the processor and the edge representing an interprocessor link.

Reference

1. W.-J. Hsu, "Fibonacci Cubes: A New Interconnection Topology," *IEEE Parallel and Distributed Systems*, Vol. 4, No. 1, Jan. 1993, pp. 3–12.

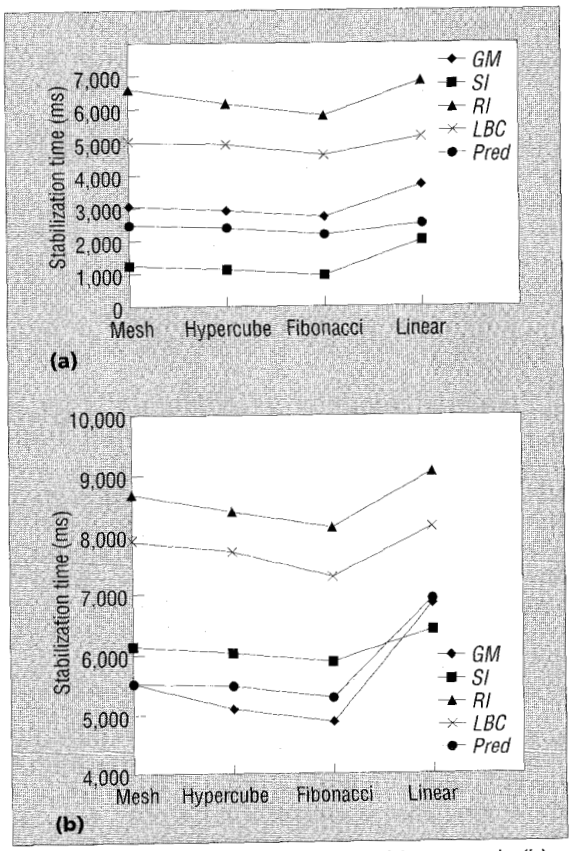


Figure 8. Stabilization times: (a) a stable network; (b) an unstable network.

Table 2. Execution times (ms) in a stable network.

Topology	GM	SI	RI	LBC	Pred	No LB	OPT
Mesh	29,683	29,460	28,784	30,084	30,428	32,529	28,162
Hypercube	29,547	28,960	28,507	29,900	30,206	32,529	28,162
Fibonacci	29,666	29,371	28,644	29,781	30,064	32,529	28,162
Linear	30,896	30,420	30,297	30,131	30,690	32,529	28,162

Table 3. Execution times (ms) in an unstable network.

Topology	GM	SI	RI	LBC	Pred	No LB	OPT
Mesh	31,962	31,573	30,788	31,075	31,419	35,498	28,162
Hypercube	31,815	31,515	30,334	30,840	31,236	35,498	28,162
Fibonacci	31,907	31,558	30,412	30,686	31,005	35,498	28,162
Linear	32,454	32,410	31,302	31,119	31,654	35,498	28,162

task-transfer process as long as there are processors whose task size is above the threshold value.

Of the last three strategies, the Pred generally has the most accurate processor-selection policy. Hence, this strategy can stabilize more quickly than the GM. The SI strategy, however, has the lowest stabilization time of the three, because the sender initiates the load-balancing only when an upper load threshold is exceeded. In the stable situation, most processors are moderately loaded and therefore do not exceed the upper load limit to trigger load-balancing activities. However, even when a network has stabilized, it might not be as effectively load-balanced as, for example, a network balanced by the Pred or the LBC strategy.

In the unstable network (Figure 8b), the average stabilization times of all strategies increase, as expected. The relative rankings of all strategies remain, except for the SI strategy. The stabilization times in the mesh, the hypercube, and the composite Fibonacci cube degrade more with this strategy than with the Pred or the GM strategy.

We have mentioned previously that the load-balancing activities in the SI strategy are sender-initiated. In the unstable situation, more processors have load levels that exceed the upper threshold, thereby increasing the load-balancing time. This situation is not reflected in the linear-array network, where the load redistribution activities in the SI occur over localized regions of the network (explained earlier), but in the Pred and the GM they occur network-wide. Therefore, the communication overheads introduced by the linearity of the structure are minimized in the case of the SI strategy.

In both stable and unstable network situations, the stabilization times remain minimal in the hypercube and the composite Fibonacci cube, and are maximum in the linear array. The results support and verify our earlier deductions of interconnection topologies with shorter Δ_{avg} and higher Θ_{avg}. The shorter average processor distance typically minimizes stabilization times directly by shortening the task migration path. The higher average node degree supports strategies relying on local-domain computational resources.

The simulation results show that topologies with larger average processor distances and lower average node connectivity introduce significant communication overheads during the load-balancing process. Because of a lack of direct links between processor nodes, task transfers need to traverse, on average, more intermediate processors before reaching the destination node. More local-domain computational resources will also be available if a processor has direct links to more nodes. The situation worsens as the load imbalance increases.

All five strategies perform best in the hypercube and the composite Fibonacci cube. The same observation applies to the performance of the application as a whole, as Table 2 (stable network) and Table 3 (unstable network) illustrate. These tables show the execution times for each load-balancing strategy in each network topology.

This research shows that varying physical parameters in an interconnection network topology significantly affect the performance of a dynamic load-balancing strategy, regardless of that strategy's approach or the network load levels. We are now working to extend our findings to develop a fault-tolerant, variable-architecture load-balancing platform.

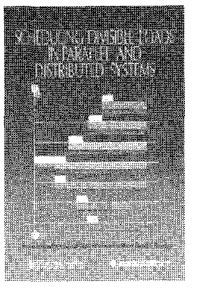
Acknowledgments

We thank Chua Chze Koon for her help in developing the simulator and collating the simulation results. We also thank the anonymous

referees for their useful comments and advice in improving this article. This work is supported by the Applied Research Fund (Grant No. RG 17/94), administered by the Ministry of Education, Singapore.

References

1. F.C.H. Lin and R.M. Keller, "The Gradient Model Load Balancing Method," *IEEE Trans. Software Eng.*, Vol. 13, No. 1, Jan. 1987, pp. 32–38.
2. D.L. Eager, E.D. Lazowska, and J. Zahorjan, "A Comparison of Receiver Initiated and Sender Initiated Adaptive Load Sharing," *Performance Evaluation*, Vol. 6, 1986, pp. 53–68.
3. F.J. Muniz and E.J. Zaluska, "Parallel Load-Balancing: An Extension to the Gradient Model," *Parallel Computing*, Vol. 21, 1995, pp. 287–301.
4. H.-C. Lin and C.S. Raghavendra, "A Dynamic Load Balancing Policy with a Central Job Dispatcher (LBC)," *IEEE Trans. Software Eng.*, Vol. 8, No. 2, Feb. 1992, pp. 148–158.
5. K.K. Goswami, M. Devarakonda, and R.K. Iyer, "Prediction-Based Dynamic Load-Sharing Heuristics," *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 6, June 1993, pp. 638–648.
6. M.A. Iqbal, J.H. Saltz, and S.H. Bokhari, "A Comparative Analysis of Static and Dynamic Load Balancing Strategies," *ACM Performance Evaluation Revision*, Vol. 11, No. 1, 1985, pp. 1040–1047.
7. M.H. Willebeek-LeMair and A.P. Reeves, "Strategies for Dynamic Load Balancing on Highly Parallel Computers," *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 9, Sept. 1993, pp. 979–993.
8. S. Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing," *IEEE Trans. Software Eng.*, Vol. 14, No. 9, Sept. 1988, pp. 1327–1341.
9. D.J. Evans and W.U.N. Butt, "Dynamic Load Balancing Using Task-Transfer Probabilities," *Parallel Computing*, Vol. 19, No. 8, Aug. 1993, pp. 897–916.
10. W.-J. Hsu, "Fibonacci Cubes: A New Interconnection Topology," *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 1, Jan. 1993, pp. 3–12.
11. O. Kremien and J. Kramer, "Methodical Analysis of Dynamic Load Balancing," *IEEE Trans. Parallel and Distributed Systems*, Vol. 3, No. 6, Nov. 1993, pp. 747–760.



Scheduling Divisible Loads in Parallel and Distributed Systems

by V. Bharadwaj, D. Ghose, V. Mani, and T. Robertazzi

Contents: The System Model • Load Distribution in Linear Networks • Load Distribution in Tree and Bus Networks • Optimality Conditions for Load Distribution • Analytical Results for Linear Networks • Optimal Sequencing and Arrangement in Single-Level Tree Networks • Asymptotic Performance Analysis: Linear Tree Networks • Efficient Utilization of Front-Ends in Linear Networks • Multi-Installment Load Distribution in Single-Level Tree Networks • Multi-Installment Load Distribution in Linear Networks • Multi-Job Load Distribution in Bus Networks

312 pages. Hardcover. August 1996. ISBN 0-8186-7521-7. Catalog # BP07521 — $40.00 Members / $50.00 List

50 YEARS OF SERVICE
IEEE COMPUTER SOCIETY
1946-1996

Phone Orders:
+1-800-CS-BOOKS
CS Online Catalog:
www.computer.org

Peter Kok Keong Loh heads the Parallel Processing Laboratory in the School of Applied Science at Nanyang Technological University, Singapore. His research interests include multiprocessor fault tolerance, parallel architectures, and parallel software. He received his B.Eng. in 1985 and MS in 1989, both in electrical engineering, from the National University of Singapore. He also obtained an MS in computer science (parallel processing) from the Victoria University of Manchester, UK, in 1992. He is a member of the IEEE. Readers can contact Loh at askkloh@ntuvax.ntu.ac.sg.

Wen Jing Hsu is a senior lecturer at Nanyang Technological University in the Division of Software Systems. He has published actively in the areas of parallel processing, algorithms, and advanced computer architectures. He received a BS in 1975, an MS in 1978, and a PhD in 1983, all in computer science, from the National Chiao Tung University. He received the General Electric Faculty Development Fund Grant in 1988 and the McDonnell Research Grant in 1990. Readers can contact Jing at aswjhsu@ntuvax.ntu.ac.sg.

Cai Wentong is a lecturer in the School of Applied Science at Nanyang Technological University. His research interests include visual-programming tools for parallel processing, parallel discrete-event simulation, cluster and heterogeneous computing, parallelizing compilers, data-parallel programming, and architecture-independent parallel computation. He received his BS in 1985 and MS in 1987 from Nankai University, People's Republic of China, and his PhD from the University of Exeter, UK, in 1990, all in computer science. Wentong joined Queen's University in Canada in 1991 as a postdoctoral research fellow. Readers can contact Wentong at aswtcai@ntuvax.ntu.ac.sg.

Nadarajah Sriskanthan is a senior lecturer in the School of Applied Science at Nanyang Technological University. He received a BS in electrical engineering from the University of London in 1972 and an MS in electronic-equipment design from the Cranfield Institute of Technology, UK, in 1979. His research interests include the development of novel parallel architectures and the applications of computer interfacing techniques. Readers can contact Sriskanthan at nil@ntuvax.ntu.ac.sg.

Readers can contact all the authors at the Division of Computing Systems, School of Applied Science, Nanyang Technological University, Nanyang Avenue, Singapore 2263, Singapore.