Strategies for Dynamic Load Balancing on Highly Parallel Computers

Marc H. Willebeek-LeMair, *Member, IEEE,* and Anthony P. Reeves, *Senior Member, IEEE*

***Abstract*— Dynamic load balancing strategies for minimizing the execution time of single applications running in parallel on multicomputer systems are discussed. Dynamic load balancing (DLB) is essential for the efficient use of highly parallel systems when solving non-uniform problems with unpredictable load estimates. With the evolution of more highly parallel systems, centralized DLB approaches which make use of a high degree of knowledge become less feasible due to the load balancing communication overhead. Five DLB strategies are presented which illustrate the tradeoff between 1) knowledge—the accuracy of each balancing decision, and 2) overhead—the amount of added processing and communication incurred by the balancing process. The *Sender (Receiver) Initiated Diffusion (SID/RID)* strategies are asynchronous schemes which only use near-neighbor information. The *Hierarchical Balancing Method (HBM)* organizes the system into a hierarchy of subsystems within which balancing is performed independently. The *Gradient Model (GM)* employs a gradient map of the proximities of underloaded processors in the system to guide the migration of tasks between overloaded and underloaded processors. Finally, the *Dimension Exchange Method (DEM)* requires a synchronization phase prior to load balancing and then balances iteratively. All five strategies have been implemented on an Intel iPSC/2 hypercube. Our results indicate that the RID approach performs well, and can most easily be scaled to support highly parallel systems.**

***Index Terms*— Distributed control, dynamic load balancing, highly parallel systems, hypercube multicomputer, multicomputer synchronization, nonuniform problems.**

I. Introduction

MULTIPROCESSOR systems have been shown to be very efficient at solving problems that can be partitioned into tasks with uniform computation and communication patterns. However, there exists a large class of nonuniform problems with uneven and unpredictable computation and communication requirements. Dynamic load balancing (DLB) schemes are needed to efficiently solve non-uniform problems on multiprocessor systems [1]. Many load balancing techniques designed to support distributed systems (e.g., Local Area Networks) have been proposed and reviewed in the literature [2]–[7]. However, only a few strategies have been designed, or are scalable, to support highly parallel multicomputer systems (e.g., tightly coupled message-passing and shared memory systems) [8]–[10], [1], [12], [13]. We are interested in dynamic load balancing schemes which seek to minimize total execution time of a single application running in parallel on a multicomputer system. To do so, an optimal tradeoff between the processing and communication overhead and the degree of knowledge used in the balancing process must be sought.

Manuscript received January 15, 1990; revised January 16, 1993. This work was supported in part by an AT&T Bell Laboratories Ph.D. Scholarship.

M. H. Willebeek-LeMair is with IBM T. J. Watson Reserch Center, Yorktwon Heights, NY 10598.

A. P. Reeves is wih the School of Electrical Engineering, Cornell University, Ithaca, NY 14853.

IEEE Log Number 9212275.

We have developed a general model for dynamic load balancing [14]. This model is organized as a four phase process: 1) processor load evaluation, 2) load balancing profitability determination, 3) task migration strategy, and 4) task selection strategy. The first and fourth phases of the model are problem dependent and purely distributed; that is, both of these phases can be executed independently on each individual processor. The second and third phases of the load balancing process can be performed in either a distributed or centralized fashion. Centralized approaches tend to be more accurate yet more time consuming and less feasible as the number of processors in the system becomes large. We are primarily interested in distributed approaches which can be scaled to support highly parallel multicomputer systems.

The tradeoff between knowledge and overhead is illustrated, by example, with five different DLB schemes. The schemes presented vary in the amount of processing and communication overhead and in the degree of knowledge used in making balancing decisions. The load balancing overhead includes the communication costs of acquiring load information and of informing processors of load migration decisions, and the processing costs of evaluating load information to determine task transfers.

Sender Initiated Diffusion (SID)[1] is a highly distributed local approach which makes use of near-neighbor load information to apportion surplus load from heavily loaded processors to underloaded neighbors in the system. Global balancing is achieved as tasks from heavily loaded neighborhoods diffuse into lightly loaded areas in the system.

Receiver Initiated Diffusion (RID) is the converse of the SID strategy, where underloaded processors requisition load from heavily loaded neighbors.

Hierarchical Balancing Method (HBM) is an asynchronous, global, approach which organizes the system into a hierarchy of subsystems. Load balancing is initiated at the lowest levels in the hierarchy with small subsets of processors and ascends to the highest level which encompasses the entire system. This scheme centralizes the balancing process at different levels of the tree with increasing degrees of knowledge at higher levels.

[1] Diffusion schemes are well known and are discussed in [15], [16].

Gradient Model (GM) [8] employs a gradient map of the proximities of underloaded processors in the system to guide the migration of tasks between overloaded and underloaded processors.

Dimension Exchange Method (DEM) [15], [17] is a global, fully synchronous, approach. Load balancing is performed in an iterative fashion by "folding" an N processor system into $\log N$ dimensions and balancing one dimension at a time.

We have categorized the issues concerning DLB strategies for highly parallel systems into the following areas: sender or receiver initiation of the balancing, size and type of balancing domains, degree of knowledge used in the decision process, aging of information in the decision process, and overhead distribution and complexity. All five schemes are compared in terms of these points. Another important issue is the effectiveness of the strategies on systems with different interconnection topologies. The HBM strategy, for example, due to its hierarchical nature, is most efficiently mapped to systems which are based on a tree or other hierarchical topology. Similarly, the DEM strategy is designed for a hypercube topology and is implemented much less efficiently on a simpler configuration such as a mesh. A final point of comparison concerns the locality of tasks. Applications comprised of tasks with a high measure of locality (local communication dependencies between tasks) would be more efficiently processed using a DLB strategy which maintains the locality of tasks. The local balancing strategies such as the SID and RID approaches do this.

All five strategies have been implemented on an Intel iPSC/2 hypercube multicomputer. Section II presents a general load balancing model upon which our strategies are based. This also includes a model of the problem decomposition and the processor loading. Section III contains an outline and complexity analysis of the five DLB strategies. The different techniques are compared in Section IV and load balancing results using sample applications are summarized in Section V.

II. A General Dynamic Load Balancing Model

The dynamic load balancing schemes we are proposing are based on a general four-phase load balancing model. A detailed description of the model is given in [14]. The four phases are described as follows.

1) *Processor Load Evaluation:* A load value is estimated for each processor in the system. These values are used as input to the load balancer to detect load imbalances and make load migration decisions.
2) *Load Balancing Profitability Determination:* The *imbalance factor* quantifies the degree of load imbalance within a processor domain. It is used as an estimate of potential speedup obtainable through load balancing and is weighed against the load balancing overhead to determine whether or not load balancing is profitable at that time.
3) *Task Migration Strategy:* Sources and destinations for task migration are determined. Sources are notified of the quantity and destination of tasks for load balancing.
4) *Task Selection Strategy:* Source processors select the most suitable tasks for efficient and effective load balancing and send them to the appropriate destinations.

The first and fourth phases of the model are application dependent and purely distributed. Both of these phases can be executed independently on each individual processor. For the purpose of this paper, we assume a simple problem characterization in which the problem is partitioned into a fixed number of tasks. All tasks are independent and may be executed on any processor in any sequence. Furthermore, due to the unpredictable nature of the task requirements, each task is estimated to require equal computation time. The initial task distribution is made based on the estimated requirements. Hence, the *Processor Load Evaluation Phase* is reduced to a simple count of the number of tasks pending execution. Similarly, the *Task Selection Strategy* is simplified since no distinction is made between tasks, and the issue of locality is ignored. For the case where tasks are created dynamically, if the arrival rate is predictable then this information can be incorporated into the load evaluation [18], if not predictable, then the potential arrival of new tasks can effectively be ignored.

Our focus is on the Profitability Determination and Task Migration phases, the second and third phases, of the load balancing process. As the program execution evolves, the inaccuracy of the task requirement estimates leads to unbalanced load distributions. The imbalance must be detected and measured (Phase 2) and an appropriate migration strategy devised to correct the imbalance (Phase 3). These two phases may be performed in either a distributed or centralized fashion. Centralized approaches tend to be more accurate since the entire system's state information is accumulated to a single point, and a high degree of knowledge is used in the decision process. However, the accumulation of information requires synchronization which incurs an overhead and a delay. This overhead may become prohibitively large for highly parallel systems and the delay may increase to a point where the information accumulated ages and loses validity. Alternatively, distributed approaches, although less accurate since they operate with less information, incur a smaller synchronization overhead.

During the Profitability Determination Phase (triggered by a processor's load estimate or timer expiration) a decision is made as to whether or not to invoke the load balancer. The load *imbalance factor* $\phi(t)$ is an estimate of the potential speedup obtainable through load balancing at time t [14]. It is defined as the difference between the maximum processor loads before and after load balancing, L_{max} and L_{bal}, respectively.

$$\phi(t) = L_{max} - L_{bal}. \tag{1}$$

A decision on whether or not to load balance is made based on the value of $\phi(t)$ relative to the balancing overhead, $L_{overhead}$, required to perform the load balancing. In general, load balancing is profitable if the savings is greater than the overhead, i.e.,

$$\phi(t) > L_{overhead}. \tag{2}$$

This may be simplified by setting $L_{overhead}$ to a constant threshold, $K_{overhead}$, rather than calculating $L_{overhead}$ as a function of the system state,

$$\phi(t) > K_{overhead}. \tag{3}$$

A further simplification is to assume that once a processor's load, L_p, drops below a preset threshold, $K_{overhead}$, any balancing will improve system performance.

$$L_p < K_{underload}. \tag{4}$$

This eliminates the need to acquire any information regarding other processor loads.

The responsibility of invoking the balancer may either be authorized to all processors in the system or only to designated processors containing the necessary information. For highly parallel systems it is desirable to distribute the responsibility to multiple points in the system. This may be accomplished by partitioning the system into independent groups of processors called *balancing domains*. The size of a balancing domain may range anywhere from a few processors to the entire system. Load balancing decisions are based solely on information pertaining to those processors within each domain. The notion of balancing domains is a way of distributing the balancing process. Furthermore, by decreasing the number of processors being considered in the balancing process, balancing domains reduce the complexity of calculating the imbalance factor as well as the complexity of phase 3, the *Load Migration Strategy*.

The role of the Task Migration Phase is first, to balance the load within each domain by identifying source and destination processor pairs and determining the amount of load to be transferred between them, and second, to balance the load across the entire system. The concept of balancing domains reduces the overhead of the balancing process, but does not ensure a balanced load for the entire system. This is accomplished by either overlapping domains, whereby excess load can diffuse from more heavily loaded domains into lightly loaded ones, or through the use of variable domains, whereby the set of processors belonging to a domain periodically changes to encompass a different subset of processors within the system. Potentially more accurate migration strategies are made possible by larger balancing domains. However, larger domains may increase the aging period of information and cause the load balancing overhead to be more unevenly distributed. These tradeoffs are illustrated by the different strategies to be discussed.

III. Dynamic Load Balancing Strategies

The following DLB strategies are designed to support highly parallel systems. For highly parallel systems, the inclination is towards distributed approaches which incur small overheads both in the communication costs for load updates, task transfers, etc., as well as in the computational costs of profitability determination, task migration decisions, etc. The five DLB strategies presented illustrate the tradeoff between 1) knowledge—the accuracy of each balancing decision, and 2) overhead—the amount of added processing and communication incurred by the balancing process. The load balancing overhead includes the communication costs of acquiring load information and of informing processors of load migration decisions, and the processing costs of evaluating load information to determine task transfers. Typically, the more information accumulated to be used in the decision process, the more accurate the decisions become. However, other factors, such as the aging of information and the rate at which the load is changing must also be considered.

A level of compromise between the degree of knowledge and the overhead of the load balancing process is sought to minimize the runtime for any given application. The DEM strategy, the only synchronized strategy, arrives at optimal load balancing decisions at the expense of high synchronization costs. Every time the load balancer is invoked the entire system is balanced. It serves as a good comparison to the other strategies which all execute asynchronously and load balance in small steps. The GM strategy incurs a relatively low overhead, but requires the fine tuning of several threshold parameters which contribute to the degree of knowledge used in the decision process. The SID and RID strategies are uniformly distributed diffusion approaches, which only use near-neighbor information in the balancing process. Finally, the HBM strategy can be thought of as an asynchronous version of the DEM approach, where balancing decisions are centralized at different levels of the hierarchy. The strategies, to be discussed, pertain to the second (Load Balancing Profitability Determination) and third (Task Migration) phases of the general load balancing model.

A. The Gradient Model (GM)

The gradient model is a demand driven approach [8]. The basic concept is that underloaded 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. The resulting effect is a form of relaxation where tasks migrating through the system are guided by the proximity gradient and gravitate towards underloaded points. The scheme is based on two threshold parameters: the *Low-Water-Mark (LWM)* and the *High-Water-Mark (HWM)*. A processor's state is considered light if its load is below the LWM, heavy if above the HWM, and moderate otherwise. A node's *proximity* is defined as the shortest distance from itself to the nearest lightly loaded node in the system. All nodes are initialized with a proximity of w_{max}, a constant equal to the diameter of the system. The proximity of a node is set to 0 if its state becomes light. All other nodes p with near-neighbors n_i compute their proximity as

$$\text{proximity}(p) = \min_i(\text{proximity}(n_i)) + 1. \tag{5}$$

A node's proximity may not exceed $w_{\max}$. A system is saturated, and does not require load balancing if all nodes report a proximity of w_{max}. If the proximity of a node changes it must notify its near-neighbors. Hence, the balancing process is initiated by lightly loaded processors reporting a proximity of 0. A gradient map of the proximities of underloaded processors in the system serves to route tasks between overloaded and underloaded processors.

Load balancing profitability determination is controlled by the LWM and HWM thresholds. In order for load balancing to take place, there must be at least one overloaded processor and one underloaded processor in the system. No measure of the degree of imbalance is made, only that one exists. This criterion is characterized by the simplified version of the load balancing profitability determination phase (3), where, given an overloaded processor p and an underloaded processor q,

$$L_p - L_q > \text{HWM} - \text{LWM}. \tag{6}$$

The proximity map is used to perform the migration phase. If a processor's state is heavy and any of its near-neighbors report a proximity less than w_{max}, then it sends a unit of its load to the neighbor of lowest proximity. Tasks are routed through the system in the direction of the nearest underloaded processors. A task continues to migrate until it reaches an underloaded processor or it reaches a node for which no neighboring nodes report a lower proximity. The scheme is illustrated in Fig. 1. In this example, there are two overloaded nodes in the system and one underloaded node. The overloaded nodes are at different proximities from the underloaded node, but both send a fraction of load, δ, in the direction of the underloaded processor. The value of δ can be determined as either a percentage of the initial load, or as a fixed number of tasks. The scheme may perform inefficiently when either too much or too little work is sent to an underloaded processor.

1) GM Complexity: The overhead incurred by the GM strategy includes the updating of the proximity map and the routing of tasks from overloaded processors to underloaded ones. The proximity map is constructed in an iterative fashion as the existence of an underloaded processor is propagated through the system. The number of messages required to update the proximity map is dependent on the interconnection network topology, the number of underloaded processors, their locations, and the order in which they become underloaded. All but one of these factors are application dependent, making it very difficult to model the complexity. Given N processors interconnected using a hypercube topology, in the worst case, an update of the gradient map, to recognize the presence of a new underloaded processor, would require,

$$C_{tot}(\text{update}) = N \log N \text{messages}. \tag{7}$$

The worst case occurs when there are no other underloaded processors in the system. Normally, the presence of other underloaded processors will halt the propagation of update messages to processors closer to the existing underloaded processors than to the new one.

The migration of tasks from overloaded to underloaded processors incurs added overhead due to the asynchronous nature of the algorithm. An overloaded processor sends a fixed portion of load in the direction of the nearest underloaded processor. Since the ultimate destination of migrating tasks is not explicitly known, intermediate processors in the migration path must be interrupted to perform the routing. Furthermore,

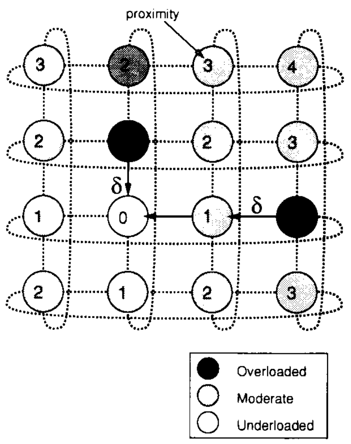


Fig. 1. The gradient model. In this example two overloaded processors release an amount of load (δ) in the direction of the nearest underloaded processor. The load migrates to the underloaded processor via the gradient map of proximity values.

the proximity map may change during a task's migration, altering its destination. This may occur since the quantity of the imbalances is not known and multiple overloaded processors may send tasks towards the same destination, creating a migration overflow and a sudden change in the proximity gradient. At the other extreme, an overloaded processor, in transferring a preset portion of load, may not send enough to solve the imbalance. Hence, the degree of information used in the balancing process may lead to inefficient migration decisions.

B. Sender Initiated Diffusion (SID)

The SID strategy is a, local, near-neighbor *diffusion* approach which employs overlapping balancing domains to achieve global balancing. A similar strategy, called Neighborhood Averaging, is proposed in [12]. The scheme is purely distributed and asynchronous. Each processor acts independently, apportioning excess load to deficient neighbors. It has been shown in [16], that for an N processor system with a total system load L unevenly distributed across the system, a diffusion approach, such as the SID strategy, will eventually cause each processor's load to converge to L/N.

Balancing is performed by each processor whenever it receives a load update message from a neighbor indicating that the neighbors load, $l_i < L_{LOW}$, where L_{LOW} is a preset threshold. Each processor is limited to load information from within its own domain, which consists of itself and its immediate neighbors. All processors inform their near-neighbors of their load levels and update this information throughout program execution. Details of the update strategy are discussed in Section III-E. The profitability of load balancing is determined by first computing the average load in the domain, $\bar{L}_p$,

$$\bar{L}_p = \frac{1}{K+1}\left(l_p + \sum_{k=1}^{K} l_k\right) \tag{8}$$

where K is the number of neighboring processors and l_k their respective loads. Next, if a processor's load exceeds the average load by a prespecified amount, $L_{threshold}$, it proceeds to implement the third phase of the load balancing process. This closely resembles the profitability determination characterized by (3).

$$L_p - \bar{L}_p > L_{threshold}. \tag{9}$$

Task migration is performed by apportioning excess load to deficient neighbors. Each neighbor k is assigned a weight h_k according to the following formula,

$$h_k = \begin{cases} \bar{L}_p - l_k, & \text{if } l_k < \bar{L}_p, \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

These weights are summed to determine the total deficiency,

$$H_p = \sum_{k=1}^{K} h_k. \tag{11}$$

Finally, the portion of processor p's excess load that is assigned to neighbor k, δ_k, is defined as

$$\delta_k = \left(l_p - \bar{L}_p\right)\frac{h_k}{H_p}. \tag{12}$$

Once the quantity of load to be migrated has been determined, the appropriate number of tasks are dispatched (note that $\delta_k \geq \delta_{MIN}$). Balancing continues throughout program execution whenever a processor's load exceeds the local average by more than a certain amount [(9)]. The scheme is illustrated in Fig. 2, where $K = 4$ and the average load in the domain, $\bar{L} = 10$. The processor in question has a surplus load of $S = 21$ and the domain deficiency is $H = 20$. The amount of load apportioned to each neighbor is represented by δ_k.

1) SID Complexity: The SID near-neighbor balancing scheme distributes the load balancing overhead uniformly across all processors. For a K-connected (K neighbors) system of N processors, a complete system load update can be expressed in terms of the number of messages sent,

$$C_{tot}(\text{update}) = KN \text{ messages}, \tag{13}$$

and the overhead incurred by each processor is

$$C_p(\text{update}) = \frac{C_{tot}(\text{update})}{N} = \frac{KN}{N} = K \text{ messages}. \tag{14}$$

The number of task transfers occurring in a given balancing iteration is dependent on a particular load distribution as well as the topology of the communication network. In the worst case, for a single task migration iteration, the communication overhead to transfer load would entail

$$C_{tot}(\text{migration}) = \frac{N}{2}K \text{ transfers} \tag{15}$$

where each transfer could require multiple messages to complete. Finally, the total number of iterations required to achieve global balancing is also dependent on the particular load distribution and the system topology. In the worst case, say for a spiked load distribution, where all the load is located on a single node, it would require $O(KN)$ transfers to balance in $O(\text{diameter}(N))$ iterations. However, these distributions rarely occur in practice and, in general, the number of required iterations is much lower.

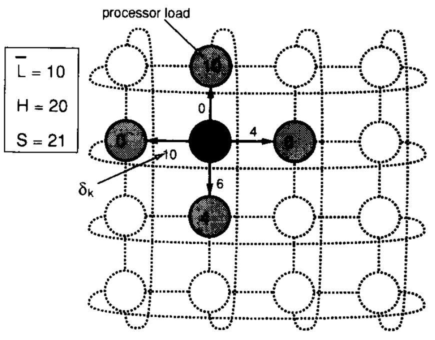


Fig. 2. An example of the SID strategy, where surplus load is apportioned to neighboring underloaded processors in the domain.

C. Receiver Initiated Diffusion (RID)

The RID strategy can be thought of as the converse of the SID strategy in that it is a receiver initiated approach as opposed to a sender initiated approach. However, besides the fact that in the RID strategy underloaded processors request load from overloaded neighbors, certain subtle differences exist between the strategies. First, the balancing process is initiated by any processor whose load drops below a prespecified threshold (L_{LOW}). Second, upon receipt of a load request, a processor will fulfill the request only up to an amount equal to half of its current load (this reduces the effect of the aging of the data upon which the request was based). Finally, in the receiver initiated approach the underloaded processors in the system take on the majority of the load balancing overhead, which can be significant when the task granularity is fine.

As with the SID strategy, each processor is limited to load information from within its own domain, which consists of itself and its immediate neighbors. All processors inform their near-neighbors of their load levels and update this information throughout program execution. Details of the update strategy are discussed in Section II-F. When a processor's load drops below the prespecified threshold L_{LOW}, the profitability of load balancing is determined by first computing the average load in the domain, $\bar{L}_p$ [(8)].

If a processor's load is *below* the average load by more than a prespecified amount, $L_{threshold}$, it proceeds to implement the third phase of the load balancing process,

$$\bar{L}_p - L_p > L_{threshold}. \tag{16}$$

Task migration is performed by requesting proportionate amounts of load from overloaded neighbors. Each neighbor k is assigned a weight h_k according to the following formula,

$$h_k = \begin{cases} l_k - \bar{L}_p, & \text{if } l_k > \bar{L}_p, \\ 0 & \text{otherwise.} \end{cases} \tag{17}$$

These weights are summed to determine the total surplus, H_p [(11)].

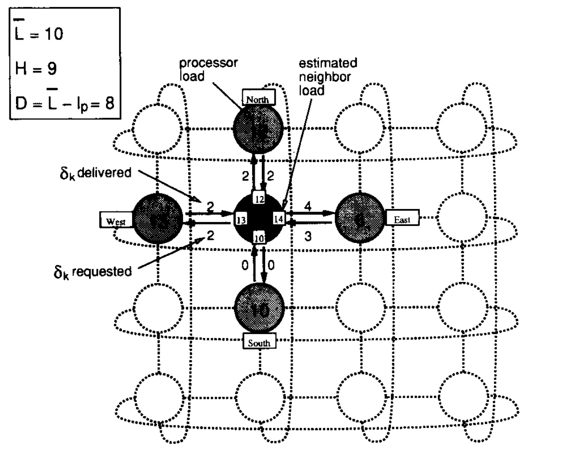


Fig. 3. An example of the RID strategy, where surplus load is requested from neighboring overloaded processors in the domain.

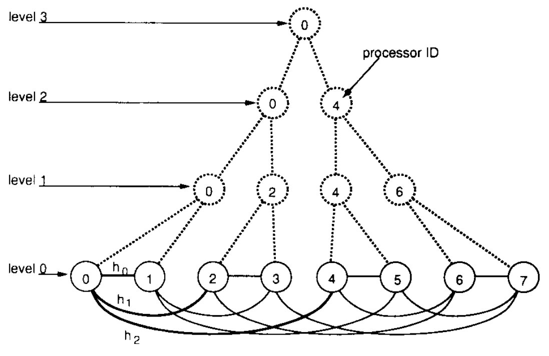


Fig. 4. Hierarchical organization of an eight-processor system with hypercube interconnections, where h_k is the connection to the neighbor at the kth level.The processor IDs at intermediate nodes in the tree represent those processors delegated to manage the balancing of corresponding lower-level domains.

Finally, the amount of load requested by processor p from neighbor k,δ_k, is computed as,

$$\delta_k = (\bar{L}_p - l_p)\frac{h_k}{H_p}. \tag{18}$$

Once the quantity of load to be migrated has been determined, load requests are sent to appropriate neighbors ($\delta_k \geq \delta_{MIN}$). However, upon receipt of a load request, a processor will fulfill the request only up to an amount equal to half of its current load. Balancing is activated whenever a processor's load drops below a prespecified threshold and there are no outstanding load requests.

The scheme is illustrated in Fig. 3, where $K = 4$ and the average load in the domain, $\bar{L} = 10$. The processor in question has a load deficiency of $D = 8$ and the domain surplus is $H = 9$. A record of pending requests is kept in order to avoid sending duplicate requests to the same neighbor. The amount of load delivered from each neighbor is represented by δ_k *delivered.* Notice that in the example, the *East* neighbor's load was estimated to be more than it actually is, and consequently it delivers less load than is requested.

1) RID Complexity: The RID near-neighbor balancing scheme distributes the load balancing overhead uniformly across all processors. For a K-connected (K neighbors) system of N processors, a complete system load update can be expressed in terms of the number of messages sent,

$$C_{tot}(\text{update}) = KN \text{ messages}, \tag{19}$$

and the overhead incurred by each processor is

$$C_p(\text{update}) = \frac{C_{tot}(\text{update})}{N} = \frac{KN}{N} = K \text{ messages}. \tag{20}$$

The RID strategy differs from its counterpart SID in the task migration phase. Here, an underloaded processor first sends out requests for load and then receives acknowledgment for each request. Hence, two additional messages are sent for each task transfer. In the worst case, for a single task migration iteration, the communication overhead to transfer load would entail

$$C_{tot}(\text{migration}) = NK\text{messages} + \frac{N}{2}K \text{ transfers} \tag{21}$$

where each transfer could require multiple messages to complete. The total number of iterations needed to achieve global balancing is application and topology dependent as with the SID approach.

D. Hierarchical Balancing Method (HBM)

The HBM strategy organizes the multicomputer system into a hierarchy of balancing domains, thereby decentralizing the balancing process. Specific processors are designated to control the balancing operations at different levels of the hierarchy. A binary-tree hierarchical organization is illustrated in Fig. 4. In this case, processors in charge of the balancing process at a level, l_i, receive load information from both lower level, l_{i-1}, domains. Global balancing is achieved by ascending the tree and balancing the load between adjacent domains at each level in the hierarchy. This procedure is asynchronous, however, where balancing is invoked within a domain whenever an imbalance is detected by the domain's designated controller. For a binary hierarchical configuration, the size of the balancing domains double from one level to the next. The tree structure minimizes the communication overhead and can be scaled to accommodate large systems.

The hierarchical balancing scheme functions asynchronously. The balancing process is triggered at different levels in the hierarchy by the receipt of load update messages indicating an imbalance between lower level domains. All load levels are initialized with each processor sending its load information up the tree. The update policy is described in Section III-F. Subtree load information is computed at intermediate nodes and propagated to the root. The update policy for intermediate nodes is the same as that of leaf processors. Load imbalances at different levels of the hierarchy are detected at intermediate nodes. If the load imbalance between two lower level domains is greater than a prespecified amount, $L_{threshold}$, then one domain is considered overloaded and the other underloaded. The absolute value of the difference between the left domain load, L_L, and the right domain load, L_R, is compared to $L_{threshold}$.

$$|L_L - L_R| > L_{threshold}. \tag{22}$$

This closely resembles the profitability determination (3). Load balancing profitability is evaluated at each level of the tree.

Task migration is controlled by intermediate nodes which, upon detecting an imbalance, notify all processors belonging to the overloaded subtree of the amount of the imbalance and at what level it occurs. Processors within the overloaded branch transfer a designated amount of their load to their "matching" neighbor in the adjacent underloaded subtree. For each processor in the left subtree at a given level, there is a corresponding processor in the right subtree, and visa-versa. Given a hypercube interconnection scheme, these processors are directly linked to one another.

In theory, the hierarchical balancing strategy guarantees that having reached a level h in the balancing hierarchy, all processors belonging to the left, level $h - 1$, subtree have equal loads and all processors within the right, level $h - 1$, subtree have equal loads. In practice, this may not quite be true due to the aging of information as well as the inaccuracy of load estimates, but, so long as the load is not fluctuating too rapidly (δ(task) $\gg$ balancing delay), the assumption is valid. Hence, as the balancing process moves up the tree to higher levels in the hierarchy, in order to correct imbalances between adjacent subtrees, all processors within an overloaded subtree may be directed to transfer an equivalent amount of load to their "matching" neighbor in the underloaded subtree. This simplifies the balancing process.

Given an imbalance, Δ_i, at a level i in the hierarchy, each processor p in the overloaded subtree is directed to transfer a load amount δ_i to its corresponding neighbor q in the underloaded subtree, where

$$\delta_i = \frac{\Delta_i}{2^i}. \tag{23}$$

For a hypercube interconnection scheme, the identity of processor p's level i neighbor, q, can be determined as a function of i and p,

$$q = p \oplus 2^{i-1} \tag{24}$$

where $\oplus$ is a bit-wise exclusive-OR operator. The hierarchical organization of an eight-processor hypercube is shown in Fig. 4.

The hierarchical scheme distributes the load balancing responsibilities to all processors in the system. It is effective for balancing local load imbalances as well as excessive global imbalances. Different imbalance thresholds can be specified at different levels of the tree. This can limit balancing domains to independent subtrees if load imbalances between subtrees do not exceed a preset threshold. Distant load transfers may then only be instituted to satisfy severe global imbalances.

1) Hierarchical Scheme Complexity: When dealing with highly parallel systems it is desirable to distribute the load balancing overhead across all processors. The load balancing overhead of the hierarchical scheme is distributed across the system, but the distribution is not uniform. The complexity of the balancing process can be analyzed in terms of the communication overhead or number of messages sent.

For a system of N processors with an embedded binary-tree communication structure (Fig. 4), a full system load update from the leaves to the root requires $N - 1$ update messages that may be completed in $\log N$ stages.

At a level i, in the hierarchical structure there are $N/2^i$ domains. If all detect imbalances, there are 2^{i-1} communicating pairs of processors per domain, yielding a maximum of

$$C_{\text{request}}(i) = C_{\text{transfer}}(i) = \left(\frac{N}{2^i}\right)(2^{i-1}) = \frac{N}{2} \tag{25}$$

load transfer request messages and an equal number of load transfers (each load transfer may involve multiple messages).

Hence, for a system of N processors, organized into a hierarchy of $\log N$ levels, a maximum of

$$C_{tot}(\text{traversal}) = N(\log N + 1) \tag{26}$$

messages are required to balance the entire system load (assuming a single message is sufficient to complete a load transfer). This may be performed with an average parallelism of $N/2$, or a maximum time complexity of $O(\log N)$.

The average cost per processor p is given by

$$\begin{aligned}\bar{C}_p(\text{traversal}) &= \frac{C_{tot}(\text{traversal})}{N} = \frac{N(\log N + 1)}{N}\\ &= \log N + 1 \text{ send and receives.}\end{aligned} \tag{27}$$

This cost is not, however, distributed evenly across all processors. The minimum cost per node (at the leaves of the tree) is given by

$$C_{min}(\text{traversal}) = 1 \text{ send} + \log N \text{ receives.} \tag{28}$$

The maximum cost per node is at the root. For the worst case,

$$\begin{aligned}C_{\max}(\text{traversal}) = &\log N \text{ receives } + N - 1 \text{ sends}\\ &+ \log N \text{ receives}\end{aligned} \tag{29}$$

where the first $\log N$ receives are update messages, the $N - 1$ sends are load transfer request messages, and the final $\log N$ receives are load transfers. Given a broadcast mechanism, the $N - 1$ sends may be reduced to $\log N$ sends—one occurring at each level in the hierarchy.

E. The Dimension Exchange Method (DEM)

The DEM strategy [15], [17] is similar to the HBM scheme in that small domains are balanced first and these then combine to form larger domains until ultimately the entire system is balanced. This differs from the HBM scheme in that it is a synchronized approach. The DEM strategy was conceptually designed for a hypercube system but may be applied to other topologies with some modifications. In the case of an N processor hypercube configuration, balancing is performed iteratively in each of the $\log N$ dimensions. All processor pairs in the first dimension, those processors whose addresses differ in only the least significant bit, balance the load between themselves. Next, all processor pairs in the second dimension balance the load between themselves, and so forth, until each

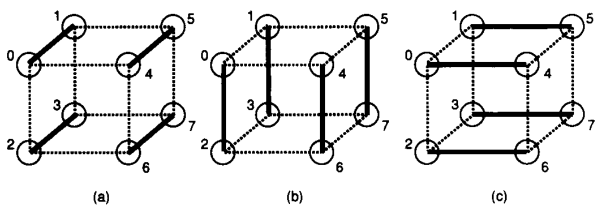


Fig. 5. DEM strategy. All processors balance in order, in each dimension. In the example shown, a three-dimensional cube of eight processors, balancing is performed between neighbors in each dimension (a), (b), and (c). Due to synchronization requirements, the balancing between each pair of nodes must follow the order shown.

processor has balanced its load with each of its neighbors. The scheme is illustrated in Fig. 5 for an eight-processor hypercube. The strategy could be extended to an $M \times M$ mesh topology by "folding" the mesh in each dimension $\lceil \log M \rceil$ times. In this case, processor pairs would no longer be directly linked to one another and communication costs would be higher.

Performing the balancing steps in a synchronized manner ensures that the entire system will achieve a balanced load. Balancing is initiated by any underloaded processor which has a load that drops below a preset threshold [(4)],

$$L_p < L_{threshold}. \tag{30}$$

This processor broadcasts a load balancing request to all other processors in the system. Global synchronization is particularly difficult in highly parallel systems where a global broadcast from a single point may be costly. Large systems equipped with special broadcast mechanisms may, however, be suited to this approach. This scheme has theoretically been shown to outperform a synchronous diffusion approach in terms of the overhead incurred to reach a uniform distribution from an unbalanced state [15]. The theoretical analysis does not include the synchronization overhead to initiate the balancing process.

1) DEM Complexity: The DEM strategy overhead is distributed uniformly across all processors in the system. Once synchronized, each processor, in an N processor system, is involved in $\log N$ balancing operations. Each of these operations includes a *load update* message, a *load transfer request* message, and a *load transfer* (which may require multiple messages). Each processor will incur 2 sends and 1 receive or 1 send and 2 receives depending on whether or not they are designated to compute the imbalance between the pair. In any case, the balancing process is evenly distributed with a total communication overhead, C_{tot},

$$C_{tot} = 3N \log N (\text{messages}). \tag{31}$$

The communication overhead incurred by each processor is,

$$C_p = 3 \log N (\text{messages}). \tag{32}$$

F. The Load Update Strategy

An important mechanism in most dynamic load balancing schemes is the load update strategy. Many dynamic load balancing strategies, like the ones proposed here, make load balancing decisions based on the load levels of a subset of processors in the system. This subset may include anywhere from a single neighbor to all processors in the system. Although the degree of knowledge may vary from one strategy to another, the quality of information governs the intelligence of the load balancing decisions. The quality of information depends on three primary factors: 1) the accuracy of processor load estimates, 2) the aging of information due to the communication latency of the interconnection network and the destination of load information, and finally, 3) the frequency of the load update messages. The first factor is application dependent and may involve a tradeoff between the quality of the estimate and the complexity of the estimation process. For example, a processor's queue length is a rough and simple load measure, a possibly more accurate yet more complex load measure might distinguish between the types of tasks within a processor's queue. The second factor is dependent on machine architecture and the load balancing strategy. The third factor, regarding the frequency of update messages, is the focus of the remainder of this section.

The interval between update messages may be computed as a function of time or as a function of the load level. If computed as a function of time, the time intervals most likely need to be adjusted for different applications. Alternatively, intervals between updates can be determined by a change in the load level, independent of the application. The updates may be sent at constant intervals, ΔL, of a processor's load, L_p. In this case, processor p must send on the order of $L_p / \Delta L$ update messages. This strategy would contribute an error margin of $\pm \Delta L$ to the accuracy of the load evaluation. Hence, smaller values of ΔL yield more accurate information, but also increase the frequency of update messages. Note also, however, that while L_p is large ($L_p \gg \Delta L$) the percentage of error in a neighbor's load information, $\Delta L / L_p$, is very small, but the error percentage increases when L_p decreases (100% when $L_p \approx \Delta L$).

Using a variable update interval, that is computed as a function of the load level, yields a constant error percentage in the load information and decreases the number of required update messages. Let the variable u be defined as the *load update factor*, such that updates are sent whenever a load, L_p, increases to $(1/u)L_p$ or drops to uL_p. If, for example, $u = 1/2$ then, a processor must send update messages whenever its load relative to that sent in its last update message has doubled or is cut in half. Processor p will send on the order of $\log_u(L_p)$ update messages. The maximum error margin of load information will be $1/u$ of the processor load L_p. The frequency of update messages will increase as the processor loads decrease and the possibility of a processor becoming idle increases. Hence, the accuracy of load information does not degrade as it becomes more critical to the load balancing process.

The load update factor expresses a tradeoff between the quality of load information and the overhead to achieve this. The larger the update factor is, the shorter the update interval becomes, and consequently, the more current or accurate the load information. Our results have indicated that the same load update factor, $u = 9/10$, is most effective for the SID and

TABLE I
SUMMARY OF COMPARISON ANALYSIS

Category	GM	SID	RID	HBM	DEM
Initiation	receiver	sender	receiver	designated	designated
Balancing Domain	variable	overlapped	overlapped	variable	variable
Knowledge	global*	local	local	global	global
Aging Period	$O(\text{diameter}(N))$	$f(u, K)$	$f(u, K)$	$f(u, N)$	constant
Overhead Distribution	uniform†	uniform	uniform	nonuniform	uniform

Recall, N is the number of processors, u is the update factor, and K is the number of near-neighbors.
* global knowledge of the locations of underloaded processors, but limited knowledge concerning the quantity of the imbalance.
† statistically uniform for a random load distribution.

RID near-neighbor schemes and that $u = 1/2$ works best for the HBM scheme. The difference is primarily due to the fact that the aging of information in the hierarchical scheme is significant in comparison to the update interval. Therefore, shorter update intervals only add communication overhead to the balancing process and sometimes create instability when out-dated messages arrive.

IV. COMPARISON ANALYSIS

The differences between the five schemes are categorized into the following areas: sender or receiver initiation of the balancing, type of balancing domain, degree of knowledge used in the decision process, aging of information in the decision process, and overhead distribution and complexity. This comparison is summarized in Table I.

A. Balancing Initiation

Our results indicate that the receiver initiated diffusion approach (RID) outperforms the sender initiated approach (SID) over the entire range of task granularities tested. We attribute this difference in performance to the following reasons:

1) *Implementation.* In theory, both approaches should yield similar results. Practical implementation issues, however, distinguish these approaches from one another. In particular, in our implementations, the SID Strategy triggers balancing operations upon the receipt of load update messages from neighbors with changing loads. The RID strategy, receives load update messages from neighbors as in the SID strategy, but balancing operations are triggered by changes in the processor's own load (i.e., when it drops below a preset threshold).
2) *Stability.* Both the SID and the RID strategies make load balancing decisions based on the load status of their near-neighbors. This load information suffers from the aging process (discussed in Section IV-D) and can at times be quite inaccurate. As a consequence, the SID strategy may transfer an excessive or insufficient amount of tasks to an "underloaded" neighbor. The RID strategy, on the other hand, may request an excessive or insufficient number of tasks from a neighbor, but the neighbor will never give out more than half of its tasks. This helps to stabilize the balancing process and reduce the effects of information aging.
3) *Overhead.* In minimizing total execution time it is beneficial to spare overloaded processors the burden of load balancing responsibilities. The extent of the overhead is dependent on the task granularity, and may become significant if tasks are small.

These results differ with those achieved in [19], where a distributed system is simulated using an analytical model. In [19] the load model assumes a continuous arrival of new tasks into the system, based on an average arrival rate. The measure used to evaluate the relative performance of different load sharing policies is the *average* service time of a task. Consequently, their objective of *load sharing*—assigning tasks to lightly loaded processors in the system, differs from our objective of *load balancing* in which we attempt to balance processor queues system wide in order to reduce the total job completion time.

Finally, the *Sender* and *Receiver* policies modeled in [19] differ from those studied here. In their *Sender* policy when a new task arrives at an "overloaded" node (a node with more than T tasks already in service or waiting for service) it probes other nodes in the system at random. A node is selected if the transfer of the task to that node would not place the node above the threshold T. If no suitable node is located after a fixed number of probes, the task is processed locally. In the Receiver policy, a node attempts to replace a task that has completed processing if there are less than T tasks remaining at that node. Nodes are probed at random (as in the Sender policy) until a suitable "overloaded" node (a node whose load exceeds T) is located. If the number of probes exceeds the predefined limit the node must wait for another task to complete before reattempting to initiate a transfer.

The study in [19] concludes that the Receiver Initiated Policy is preferable to the Sender Initiated Policy at high system loads when the transfer of tasks under the two strategies are comparable. This is understandable since in a heavily loaded system there will be fewer "underloaded" nodes that are hard to find and a Receiver initiated approach would be more effective. However, if the cost of a task transfer under the receiver initiated policy is significantly greater than under the Sender-initiated policy (due to the added cost of transferring an executing task[2]), then the Sender-initiated policy is preferable.

Our load model assumes a fixed number of tasks all present in the system at initialization time, and performance is measured as the total time required to complete all tasks. Second, the distributed system they assume consists of identical nodes connected by a local area broadcast channel (e.g., an Ethernet).

[2] In our load model there is no time-sharing of tasks and only nonexecuting tasks are considered for transfer.

Hence, communication between any two nodes in the system is of equal cost, and a random probing of any node for load balancing purposes as opposed to near-neighbor nodes makes more sense.

The GM and DEM strategies are also receiver initiated, but migration decisions are not the sole responsibility of the receiver. In the DEM scheme, once the balancer is invoked, migration decisions are made synchronously by designated processors. The GM scheme is slightly more difficult to characterize since underloaded processors (receivers) alert the system of their presence, but no explicit request is made to any particular overloaded processor (sender). Senders simply release tasks into the system in the presence of an underloaded node. Finally, in the HBM scheme migration decisions are made by designated processors in the system.

B. Balancing Domains

The notion of balancing domains was introduced in Section II-B. The use of balancing domains is a means of decentralizing the balancing process and reducing its complexity. Two types of domains exist; overlapping domains which achieve global balancing through the process of diffusion and variable domains which change shape and/or members in subsequent balancing iterations. It has been shown in [11], where they refer to the balancing domains as *buddy sets*, that for a hypercube system using overlapping domains, there exists a maximum size domain beyond which the balancing process no longer benefits by using larger domains. The SID and RID approaches employ overlapping domains while all three other approaches use variable domains. The balancing domains in the HBM strategy vary to include a larger subset of processors at higher levels in the hierarchy. The same is true for the DEM strategy where each dimension is balanced in turn. Finally, the GM domains vary according to the location of the nearest underloaded processor.

C. Degree of Knowledge

The degree of global knowledge, also referred to as *information dependency* [3], used in the balancing process is critical to the accuracy of balancing decisions. The more knowledge available in the decision process the more effectively the balancer can correct imbalances in the global load distribution. The SID and RID strategies only make use of a small degree of knowledge (load levels of K neighbors) in each balancing decision. Both the HBM and DEM strategies use only a small degrees of knowledge in each balancing step, but some additional knowledge is implicitly known. The HBM strategy is structured in such a way that, while the technique is asynchronous, lower level domains will balance themselves before upper level domains when imbalances exist. Therefore, some knowledge concerning lower level domains' load distributions may be assumed, increasing the inherent degree of knowledge used in higher level balancing decisions. The same is true for the DEM approach, but for different reasons. Here, the balancing process is explicitly synchronized forcing different level domains to balance in order. The GM scheme is perhaps the most complex to evaluate. This scheme uses a relatively large degree of knowledge, but in an indirect way via the proximity gradient map. Each processor has certain global knowledge concerning the locations of underloaded processors. This knowledge is not very informative, however, since it does not relate the exact locations of underloaded processors, nor does it give a quantitative measure of the imbalances that exist.

D. Aging of Information

The accuracy of the information used by the load balancer is vital to its effectiveness. Three of the four strategies described make use of a periodic update strategy (Section III-E). This update strategy is critical to the accuracy of load information in terms of the aging period. The *aging of information* specifically refers to the length of the delay from the time of load information determination to the time it is used in making balancing decisions. This delay is particularly critical when the load levels are changing at a rapid rate and the load information is only valid for a short period of time. Aside from the update interval, $u(t)$, the delay depends both on the system communication latency as well as on the amount of information being acquired. For the SID, RID, and HBM strategies the aging period depends primarily on the length of the update interval, $u(t)$. For the RID and SID strategies the aging period is also affected by the number of processors per domain, $O(K)$. The HBM strategy aging period depends on the hierarchical organization, including the number of levels in the hierarchy as well as on the number of processors per branch (e.g., $O(\log N)$ for a binary-tree organization). The aging period of the DEM strategy, because it operates synchronously, is constant, while that of the GM scheme is $O(\text{diameter}(N))$, where diameter(N) is the maximum number of hops between any two processors in the system.

E. Overhead Distribution and Complexity

It is desirable to both minimize the load balancing overhead as well as to distribute it evenly across all processors in the system. This eliminates any bottlenecks in the balancing process and increments in the overhead will not severely impact system performance. Furthermore, the balancing overhead should be scalable to support large systems. Both the SID and RID strategies achieve a uniform overhead distribution that is independent of N, but increases instead as $O(K)$, the number of neighbors. The RID strategy, however, requires two more messages per task transfer. The HBM scheme also distributes the load balancing overhead, but some processors incur a larger portion than others. For a binary tree organization, the disparity in the overhead distribution is $O(N/\log N)$, or $1:3$ given a broadcast mechanism [see (28) and (29)]. Nonetheless, the average overhead per processor increases as $O(\log N)$. For the DEM strategy some synchronization mechanism is required once the load balancer is invoked. The overhead of the GM scheme is difficult to measure. In setting up the gradient map each processor in the GM scheme may need to update its *proximity* $O(N)$ times. Furthermore, in the GM scheme, the processors in the path of migration incur additional overhead in forwarding tasks to their destinations. Since these destinations

TABLE II
PARAMETERS USED FOR DLB STRATEGY IMPLEMENTATIONS

GM	RID	SID	HBM	DEM
LWM = $1 + \frac{L_{INIT}}{10}$	$L_{LOW} = 1 + \frac{L_{INIT}}{10}$	$L_{LOW} = \infty$	$L_{threshold} = 1000x2^i$	$L_{LOW} = 1$
HWM = LWM x 2	$\delta_{MIN} = 1$	$\delta_{MIN} = 1$	$\delta_{MIN} = 1$	$\delta_{MIN} = 1$
$\delta = 1$	$u = \frac{9}{10}$	$u = \frac{9}{10}$	$u = \frac{1}{2}$	
max_hops = log N	$L_{threshold} = 1$	$L_{threshold} = 1$		

are not fixed, unless a limit is put on the number of hops a task is permitted to travel, tasks may continue to migrate through the system indefinitely.

V. IMPLEMENTATION RESULTS

All five schemes were implemented on an Intel iPSC/2 hypercube. The strategies were first tested and analyzed using synthetic data consisting of a set of tasks with random computational requirements executed as busy loops. After observing the strategies performances at balancing the artificial application, they were tested with a branch-and-bound Job Scheduling problem.

The parameters used in implementing the various strategies are shown in Table II. No effort was made to fine tune these parameters for each independent run. Reasonable values were used which tended to produce the best results for the majority of cases. Improved performance can be obtained for each of the strategies by adjusting these parameters with *a priori* knowledge of the application characteristics. This was not assumed to be the case.

In implementing the load balancing strategies on the iPSC/2 hypercube, some minor adaptations have been made to some of the algorithms. For the GM scheme, each task is assigned a hop counter to limit the distance (and time) that a task is permitted to travel. This is done to avoid message traffic congestion and stabilize the migration process. For the DEM strategy, the balancer is invoked by a global broadcast from any processor that becomes underloaded. This, however, presents a problem when more than one processor requests balancing simultaneously, since request messages may get queued for receipt while another request is being serviced. Queued requests become obsolete once the balancer is invoked. Hence, in order to distinguish between new and old requests each balancing sequence is given an identification number. If a pending request has an ID number that is less than or equal to the last request's ID, it is ignored. Each processor updates the current request number whenever a balancing operation is executed.

A. Load Model for the Artificially Generated Tasks

The artificial application is practical for analysis since both the size and quantity of the tasks being processed can be predetermined. The total problem load, L, is distributed (with a uniform random distribution) across all processors in blocks of size $h = \omega\psi$, where ω and ψ are defined as the task weight and the loop size, respectively. On a system of N processors each processor's load equals hl_i, where l_i is a random value in the range, $0 < l_i < 2L/hN$. The range is selected to yield

TABLE III
SAMPLE PARAMETER SET FOR ARTIFICIALLY GENERATED TASKS

grain size	task weight	loop size	total load
g	ω	ψ	L(10^8 loops)
10	50	20000	8.10
20	100	10000	8.24
50	250	4000	8.12
100	500	2000	7.97
200	1000	1000	8.15
500	2500	400	8.08
1000	5000	200	8.07

a total load of L, represented as,

$$L = h \sum_{i=0}^{N-1} l_i. \tag{33}$$

Each processor's load, hl_i, is further partitioned into g tasks of random sizes (uniform distribution), τ_j.

The block size, $h = \omega\psi$, is held constant. However, the ratio, ω/ψ is varied to change the granularity, g, while keeping the range of τ_j the same. Each processors load can be written as,

$$hl_i \approx \psi \sum_{j=1}^{g} \tau_j. \tag{34}$$

The task sizes, generated randomly (uniform distribution) on each processor, can be represented by an average task size, $\bar{\tau}$. Equation (34) can then be rewritten as,

$$\begin{aligned} hl_i &\approx \psi g \bar{\tau} \\ \omega\psi l_i &\approx \psi g \bar{\tau}. \end{aligned} \tag{35}$$

Since l_i is a constant, in order to keep the range of τ_j constant, independent of g, a change in the value of g must be counter balanced by ω. Hence, an increase in g results in a directly proportional increase in ω. This in turn results in an inversely proportional decrease in ψ due to the relation, $h = \omega\psi$, where h is constant. Also, in order to satisfy (34), the task sizes, τ_j, must be scaled to within the range, $0 < \tau_j < 2\omega l_i/g$.

In summary, the block size, h, is chosen to scale the problem size, and the ratio ω/g is held constant to fix the task size range (smaller range values improve the load estimate) for all values of g. For example, given the following values, $h = 10^6$ and $\omega/g = 5$, a sample set of parameters are shown in Table III. For an average processor load, $\bar{l}_i = 25$, on an $N = 32$ processor system, the set of parameters shown in the table would produce a random load distribution with a statistical sum of 8 x 10^8 loops. The actual total loads produced are included in Table III.

TABLE IV
RESULTS FOR ARTIFICIAL LOADS OF GRANULARITY 100 ON A 32 NODE iPSC/2

Run	Time(s)								
	optimal	NOBAL	DEM	RID		SID		HBM	GM
				$L_{LOW} = \alpha$	$L_{LOW} = \infty$	$L_{LOW} = \alpha$	$L_{LOW} = \infty$		
1	35	66	47	47	48	61	52	48	53
2	35	74	47	48	48	68	56	48	56
3	31	62	41	42	43	55	49	43	50
4	26	64	34	35	34	40	39	34	40
5	36	62	48	48	48	57	53	48	56
6	34	66	46	46	46	48	49	46	51
7	32	66	43	43	43	53	52	43	50
8	37	68	49	50	50	59	56	51	54
9	29	61	39	39	39	42	42	39	59
10	32	74	43	43	43	51	50	43	49
$\bar{T}$	32.8	66.3	43.7	44.1	44.2	53.4	49.8	44.3	51.8
Π	1.00	0.00	0.67	0.65	0.65	0.38	0.49	0.65	0.43
δ_{total}	N.A.	0	645	2862	2820	3952	4155	6694	5324
speedup	2.02	1.00	1.52	1.50	1.50	1.24	1.33	1.50	1.28

$\alpha = 1 + \frac{L_{INIT}}{10}$

$\bar{T}$ is the average time of the 10 runs measured.

Π is a measure of the load balancing effectiveness.

δ_{total} is the average total tasks transferred over the 10 runs measured.

speedup is measured as the improved speedup over the case of no load balancing (NOBAL).

B. Performance Results for the Artificially Generated Tasks

The first performance measure used to compare the effectiveness of the various load balancing strategies is made independent of system size and task granularity. All five load balancing strategies were tested on ten different runs of uniform random load distributions with a granularity of 100 tasks per processor on a 32 processor iPSC/2. The problem size was chosen as L = 8.0 x 10^8 loops, and a loop is approximately 1.3 μs. Tasks are divided into blocks of 100 loops, and messages are checked every 100 loops (130 μs). The experimental results are presented in Table IV.

Timing measurements were obtained for the RID and SID strategies using two different L_{LOW} threshold values. These results indicate that the RID strategy was not very sensitive to the threshold value while the SID strategy was. For the SID strategy, the L_{LOW} threshold must be set high since waiting until "the last minute" before adjusting load imbalances leads to processor idle time and poor performance. This is attributable to the fundamental difference between the two strategies, which is that in the RID strategy the underloaded processors are themselves responsible for obtaining work, while in the SID strategy underloaded processors must rely on neighbors to obtain work for them. Consequently, when a processor becomes underloaded in the RID scheme it will continue to perform work in the form of load balancing operations, while in the SID scheme an underloaded processor remains idle until a neighbor sends it work. A measure used to determine the effectiveness of the load balancing strategies is the normalized performance Π which takes into account the initial level of load imbalance as well as the load balancing overhead,

$$\Pi = \frac{T_{nobal} - T_{bal}}{T_{nobal} - T_{op}} \tag{36}$$

where the *optimal* time, T_{op}, is the time to complete the work on a uniprocessor divided by the number of processors in the multiprocessor system (T_{uni}/N), T_{nobal} is the time to complete the work on a multiprocessor system without load balancing, and T_{bal} is the time to complete the work on a multiprocessor system with load balancing. When the load balancing time approaches the optimal time ($T_{bal} \rightarrow T_{op}$), then the normalized performance approaches one ($\Pi \rightarrow 1$). If the load balancing is poor and does not improve much over the case without load balancing ($T_{bal} \rightarrow T_{nobal}$) then the normalized performance approaches zero ($\Pi \rightarrow 0$).

An average Π value was computed for each load balancing strategy over the ten runs. This measure is included in Table IV. The DEM, RID, and HBM strategies all performed comparably, while the SID and GM strategies performed less effectively. Note, however, that although the DEM, RID, and HBM strategies yielded good performance, the number of task transfers enacted to achieve the load balance varied dramatically. The HBM required 10 times more transfers than the DEM strategy. This variation has a significant impact on performance when the granularity of tasks is finer and many more transfers are required. All strategies, however, improved the speedup over the case with no load balancing (see Table IV).

The effect of system size and task granularity on the performance of each DLB strategy for the artificial data have been investigated. A graph of speedup versus system size is shown in Fig. 6. Each point in the graph represents the mean value taken over ten different uniform random load distributions. The problem size was scaled to match the number of processors in the system, L = 2.5 x 10^7N loops. As before, tasks are divided into blocks of 100 loops, and messages are checked every 100 loops (130 μs). The granularity of the problem was g = 100 tasks per processor.

The random load distributions affect the total load and the load imbalance for different size systems, and consequently the optimal speedup attainable. The GM and SID strategies

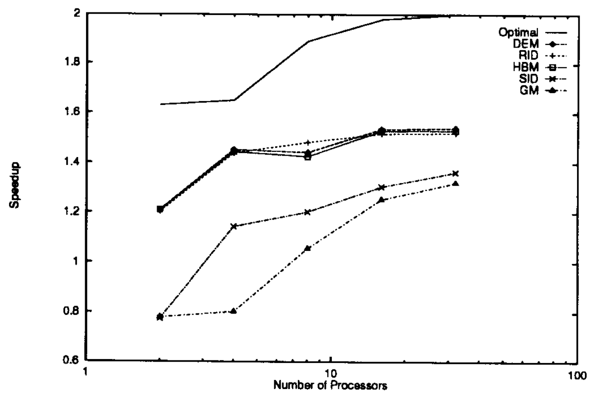


Fig. 6. Performance of different size systems at executing artificially generated random loads, with a granularity of 100 tasks per processor. The problem size is scaled to match the system size, and the performance is measured in terms of speedup over the case where no load balancing is used.

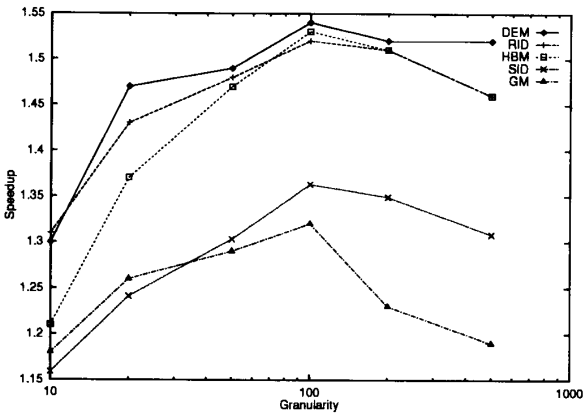


Fig. 7. Performance of a 32 processor system at executing different grain sizes of artificially generated random loads. Performance is measured in terms of speedup over the case where no load balancing is used.

were ineffective for small system sizes, but improved as the number of processors increased. From the graph we can infer an asymptotic trend in which the performance of the SID and GM strategies approach that of the other strategies for very large systems.

A graph of speedup versus granularity (tasks per processor) for a 32 processor iPSC/2 is shown in Fig. 7. Once again, each point in the graph represents the mean value taken over ten different random load distributions. The granularities selected, correspond to those shown in Table III. The total load for each individual data set was chosen to be $\approx 8 \text{ x } 10^8$ loops. An analysis of the results indicated that between 20 and 25% of the processing time was spent on load balancing. Since the optimal speedup attainable is 2, this indicates that the balancers are actually achieving near optimal load distributions, but the performance is deteriorated by the load balancing overhead.

From the graph in Fig. 7, it is evident that the DEM strategy outperforms all other strategies. This is expected since the size of the system being used is relatively small (32 processors) and the synchronization overhead is small. All strategies perform the best for the midrange of granularities, specifically when $g = 100$. Smaller grains are less efficient to transfer (only one task is transferred per message), and larger grains make it more difficult to load balance since tasks cannot be partitioned. The HBM strategy's performance degrades for small granularities due to the aging of information and the instability of the task migration protocol. Out-dated load update messages may arrive when the task execution time is small relative to the aging period of load information. The GM strategy deteriorates for finer granularities, since there is no measure of the quantity of the imbalance between processors and a fixed number of tasks are transferred in all cases. For this particular problem, the amount of load transferred when $g = 100$ is best suited to the imbalance situation that exists. Finer tuning of the load transfer amount as well as the HWM and LWM thresholds of the GM strategy may improve the performance for each case.

The RID strategy outperforms its counterpart, the SID strategy. This is attributable to several reasons: 1) subtle differences in the strategy implementations, particularly how the RID triggers load balancing operation, 2) although the balancing overhead is slightly greater for the RID strategy, the underloaded, rather than the overloaded, processors are burdened with the overhead (this can be significant when the task size is small), and 3) the added overhead, due to the additional messages in the migration protocol, improves the stability of the balancing process. The stability is improved since underloaded processors only get as many tasks as they request, and overloaded processors have a final determination in how many tasks they choose to send. Furthermore, underloaded processors do not send out more requests until all requests from previous iterations are answered.

C. The Branch-and-Bound Job Scheduling Application

Branch-and-bound (b&b) is a well-known technique for solving combinatorial search problems. The basic scheme is to reduce the problem search space by dynamically pruning unsearched areas which cannot yield better results than solutions already found. Branching is performed by recursively partitioning the problem into subproblems. A lower bound is computed for each subproblem to determine whether or not further exploration of the subproblem is worthwhile. Branching is typically performed using either a depth-first or best-first search approach.

A b&b algorithm was implemented to solve the *Job Scheduling Problem* [20]. The Job Scheduling problem is described as follows: Given a set of J jobs to be run on a single processor, each with a processing time e_j and a due date d_j, an optimal schedule is sought which minimizes the total tardiness of the set. The tardiness is expressed as

$$T = \sum_{j=1}^{J} (\max(0, X_j - d_j)) \qquad (37)$$

where X_j is the completion time of job j. No credit or penalty is given to jobs completed before their deadlines.

A sequential b&b approach to solving the Job Scheduling problem is performed by searching the solution space (depth-first or best-first) and maintaining a record of the partial

TABLE V
PARAMETER SET FOR BRANCH-AND-BOUND PROBLEM OF DIMENSION $n = 11$

task dimension m	task size $m!$	number of tasks s
6	720	56440
7	5040	7920
8	40320	990
9	362880	110

tardinesses of all partially completed schedules (branches). The minimum tardiness of all fully completed schedules is also maintained throughout the search. If a branch's partial tardiness exceeds the recorded minimum tardiness, the branch may be disregarded from further consideration. The pruning of branches can greatly reduce the search space.

The parallel b&b approach that we have implemented to solve the Job Scheduling Problem is a straightforward parallelization of the sequential approach with a minor modification. On an N processor system, the search space is divided into N parts and each processor explores its own branch. However, in order to make a fair comparison between the different load balancing strategies, a predetermined tardiness threshold is specified and all solutions satisfying this requirement are found. This modification ensures that, regardless of which strategy is used, the number of nodes searched remains constant. Without this modification, the order in which branches are searched (affected by the load balancing strategy) may alter the total problem size since early detection of a good solution can quickly prune subsequent branches. A similar approach was used by Rao and Kumar in their analysis of parallel depth-first search [21], [22].

The load distribution for the b&b Job Scheduling Problem can quickly become unbalanced since certain branches may be pruned quickly while others will be searched exhaustively. Consequently, to perform efficiently, the load must be balanced dynamically during program execution.

Our primary focus in implementing the b&b algorithm on a hypercube is on the dynamic load balancing process. The b&b problem produces tasks (branches) of varying load requirements that are easily transferred between processors. Hence, the application fits the load model assumptions made earlier. The problem search space for a b&b problem of n dimensions can be partitioned into a set of $s = n!/m!$ tasks, of size $m!$ search steps, where $n \geq m \geq 1$. A search step takes on the order of 150 μs to complete and messages are checked after each search step. The decomposition of a problem of dimension 11 into tasks of sizes $m! = 6!, 7!, 8!,$and $9!$ is summarized in Table V.

The results obtained for the b&b application have been plotted in Fig. 8. Each point in the graph represents the mean value taken over five different schedule requirements where $n = 11$. The granularities tested correspond to the parameters shown in Table V. In general, the results for the b&b application exhibit the same pattern as the graph obtained for the artificial tasks. However, the potential load imbalance for the b&b application is much greater than that of the artificial application, depending on how early task branches are pruned. The DEM and RID strategies outperformed all others. The HBM scheme achieved comparable performance for coarse granularity, but had complications with message overflows for the finer grain sizes. The grain sizes span a larger range than the artificial tasks do. The performance of all strategies suffers for very fine grain sizes. This is due to the fact that each task is transferred independently, and the communication cost of transferring the small tasks is comparable to its computation time. A block transfer approach would enhance the performance.

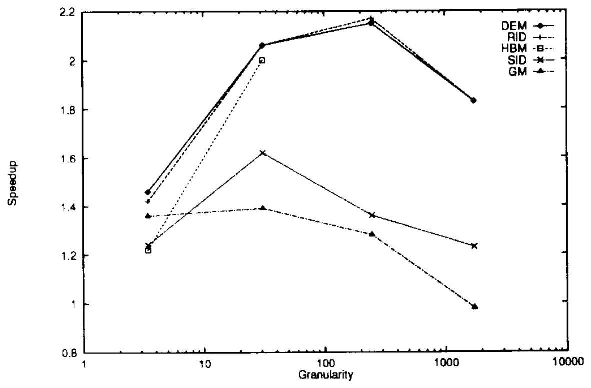


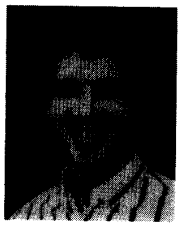
Fig. 8. Performance of a 32 processor system at executing different grain sizes of a branch and bound Job Scheduling Problem of dimension 11. Performance is measured over the case where no load balancing is used.

VI. CONCLUSION

Five dynamic load balancing strategies designed to support highly parallel systems have been presented and compared. The different strategies exemplify some of the main issues and tradeoffs that exist in dynamic load balancing, specifically in reference to highly parallel systems. Two major issues, that of load balancing overhead and the degree of knowledge used in balancing decisions were discussed. Also considered were, the concept of balancing domains, the aging of information, and the form of balancing initiation. Of the five strategies proposed, the DEM strategy tended to outperform the rest for all granularities. The efficiency of the DEM and the HBM strategies, depends heavily on the system interconnection topology. The hypercube topology is ideally suited to match these two strategies communication dependencies. Furthermore, the system sizes tested were very small in the context of highly parallel systems. The overhead of synchronization costs [scale as $O(N \log N)$] for the DEM approach and the aging period and nonuniform overhead distributions of the HBM approach may deteriorate their performance when the number of processors is large (1000 processors). The RID strategy, on the other hand, is easily ported to simpler topologies, and can scale gracefully for larger systems. Finally, for a wider variety of applications, exhibiting local communication dependencies between tasks, the RID scheme is able to maintain task locality. Therefore, since its performance was shown to be comparable to those of the DEM and HBM approaches, the RID strategy may be best suited for a broader range of systems supporting a large variety of applications.

REFERENCES

[1] M. Willebeek-LeMair and A. P. Reeves, "Region growing on a hypercube multiprocessor," in *Proc. 3rd Conf. Hypercube Concurrent Comput. and Appl.,* 1988, pp. 1033–1042.
[2] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Trans. Software Eng.,* vol. 14, no. 2, pp. 141–154, Feb. 1988.
[3] Y.-T. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Trans. Comput.,* vol. C-34, pp. 204–217, Mar. 1985.
[4] M. J. Berger and S. H. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Trans. Comput.,* vol. C-36, pp. 570–580, May 1987.
[5] G. C. Fox, "A review of automatic load balancing and decomposition methods for the hypercube," California Institute of Technology, C3P-385, Nov. 1986.
[6] K. Ramamritham, J. A. Stankovic, and W. Zhao, "Distributed scheduling of tasks with deadlines and resource requirements," *IEEE Trans. Comput.,* pp. 1110–1123, Aug. 1989.
[7] K. M. Baumgartner, R. M. Kling, and B. W. Wah, "Implementation of GAMMON: An efficient load balancing strategy for a local computer system," in *Proc. 1989 Int. Conf. Parallel Processing,* vol. 2, Aug. 1989, pp. 77–80.
[8] F. C. H. Lin and R. M. Keller, "The gradient model load balancing method," *IEEE Tran. Software Eng.,* vol. 13, no. 1, pp. 32–38, Jan. 1987.
[9] L. V. Kale "Comparing the performance of two dynamic load distribution methods," in *Proc. 1988 Int. Conf. Parallel Processing,* Aug. 1988, pp. 8–12.
[10] J. Hong, X. Tan, and M. Chen, "From local to global: An analysis of nearest neighbor balancing on hypercube," in *Proc. Third Conf. Hypercube Concurrent Comput. and Appl.,* Jan. 1988.
[11] K. G. Shin and Y.-C. Chang, "Load sharing in distributed real-time systems with state-change broadcasts," *IEEE Trans. Comput.,* pp. 1124–1142, Aug. 1989.
[12] V. A. Saletore, "A distrubuted and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks," in *Proc. Fifth Distributed Memory Comput. Conf.,* Apr. 1990, pp. 995–999.
[13] W. Shu and L. V. Kale, "A dynamic scheduling strategy for the Chare-kernel system," in *Proc. ACM Supercomput. Conf.,* 1989, pp. 389–398.
[14] M. Willebeek-LeMair and A. P. Reeves, "A general dynamic load balancing model for parallel computers," Tech. Rep. EE-CEG-89-1, Cornell School of Electrical Engineering, 1989.
[15] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel and Distributed Comput.,* vol. 7:279-301, October, 1989.
[16] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods.* Englewood Cliffs, NJ: Prentice-Hall, 1989.
[17] K. M. Dragon and J. L. Gustafson, "A low-cost hypercube load balance algorithm," in *Proc. Fourth Conf. Hypercubes, Concurrent Comput. and Appl.,* 1989, pp. 583–590.
[18] M. Hailperin, "Load balancing for massivelly-parallel soft-real-time systems," in *Proc. 2nd Symp. Frontiers of Massively Parallel Computation,* 1988, pp. 159–163.
[19] D. E. Eager, E. D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," *Perform. Eval.,* vol. 6, pp. 53–68, 1986, North-Holland.
[20] R. P. Pargas and E. D. Wooster, "Branch-and-bound algorithms on a hypercube," in *Proc. Third Conf. Hypercube Concurrent Comput. and Appl.,* Jan. 1988.
[21] V. N. Rao and V. Kumar "Parallel depth first search. Part I. Implementation," *Int. J. Parallel Programming,* vol. 16, no. 6, 1987.
[22] ——, "Parallel depth first search. Part II. Analysis," *Int. J. Parallel Programming,* vol. 16, no. 6, 1987.



Marc H. Willebeek-LeMair (S'84–M'90) received the B.S. degree in computer and electrical engineering from George Mason University, Fairfax, VA, in 1985, and the M.S. and Ph.D. degrees from the School of Electrical Engineering, Cornell University, Ithaca, NY, in 1988 and 1990, respectively.

Since 1990 he has been working in the High Bandwidth Systems Laboratory of the IBM T. J. Watson Research Center. His research interests include parallel processing, high bandwidth communications, computer architecture, and interconnection networks.

Dr. Willebeek-LeMair is a member of the IEEE Computer Society, the IEEE Communications Society, Alpha Xi, and Eta Kappa Nu.

Anthony P. Reeves (M'76–SM'84) received the B.Sc. degree (honors first class) in electronics and the Ph.D. degree from the University of Kent, Canterbury, U.K., in 1970 and 1973, respectively.

He is currently an Associate Professor in the School of Electrical Engineering, Cornell University, Ithaca, NY. Previously, from 1976 to 1982, he was an Assistant Professor in the School of Electrical Engineering, Purdue University. He has held visiting faculty positions at the University of Wisconsin, Madison; McGill University, Montreal, P.Q., Canada; and Pavia University, Italy. From 1987 to 1988 he was a member of the faculty of the Department of Computer Science, University of Illinois at Urbana–Champaign. His current research interests include parallel processing and computer vision. He developed the high level programming language, called Parallel Pascal, for NASA's Massively Parallel Processor and he is now working on dynamically adaptable programming environments for multicomputer systems. Vision research is centered on the analysis of multiframe image sequences and on the identification of three-dimensional objects for robot vision applications.

Dr. Reeves is a member of the Association for Computing Machinery, Sigma Xi, and Eta Kappa Nu.