THE PEBBLE CRUNCHING MODEL FOR LOAD BALANCING IN CONCURRENT HYPERCUBE ENSEMBLES

C^3P-610

Sandeep Gulati† *Jacob Barhen*‡ *S. Sitharama Iyengar*†

‡ Jet Propulsion Laboratory, Caltech
4800 Oak Grove Drive
Pasadena, California 91109

† Department of Computer Science
Louisiana State University
Baton Rouge, LA 70803

ABSTRACT

The successful development of fifth generation systems require enormous computational capability and flexibility necessitating the ability to achieve operational responses in hard real-time through optimal resource utilization. This entails dynamically balancing the computational load among all the processing nodes in the system. We propose a graph-theoretic, receiver-initiated, distributed protocol for dynamic load balancing protocol in large-scale hypercube ensembles. Using attributed hypergraphs as the primary data structure for constraint modeling and dynamic optimization, we consider systems running precedence-constrained heterogeneous tasks. Fault Tolerance is ensured by incorporating an integrity check for the decision nodes and their subsequent reelection if needed. Simulation studies are used to evaluate the performance of the algorithm.

I. INTRODUCTION

Real time optimization of overall performance of a distributed processing system requires that, the tasks being executed be uniformly distributed amongst the various processing nodes, in a manner which maximizes resource utilization to enhance the total throughput of the system. *Load balancing* then, is a " distributed decision process " [9] which using a local view of the global system state, arbitrates on the assignment of the system's resources to the tasks requesting them. In general, given a job load composed of modules with interlying dependencies to be executed on a multiprocessor configuration with prefixed interconnection network, determine an assignment pattern, or, a mapping function for shuffling tasks between the processors, such that total execution time of the job is minimized by avoiding under-utilized processors. The difficulty here lies in the conflict of constraints over a configuration space which grows exponentially with the number of tasks.

Determination of feasible assignment patterns for a given system may be *static*, as discussed by Barhen [1], Chou and Kohler [3], Livny [9] and Tantawi and Towsley [17] or *dynamic* Eager et al [4], Lin and Keller [8], Stankoviv and Sidhu [16]. If the mapping is static then the tasks and their dependencies are known apriori and can be mapped onto the network nodes before the computation begins. Once assigned to a particular processor the tasks are bound to it during their entire lifetime. On the other hand in dynamic assignment the computation is modeled by a dynamically created task precedence graph and the performance depends upon the process migration mechanism and the size of information domain analysed for load dispersal. Hereafter we focus on the dynamic load balancing.

Distributed systems may adopt either *sender-initiated* or *receiver-initiated* strategies for dynamic load balancing. In systems using sender-initiated requests, the overloaded nodes search for underloaded nodes to which some of their excess load may be transferred while in the latter the situation is reversed and underutilized nodes search for congested nodes from which load may be transferred to enhance the throughput by preventing processor inactivity due to lack of task availability. Analytical models and simulations have shown [4] sender-initiated strategies outperform receiver-initiated strategies at light to moderate system loads while receiver-initiated strategies are preferable at high system loads, assuming that the process migration cost under the two strategies are comparable. Receiver initiated policies require the transfer of executing partially completed tasks, thus incurring substantial process migration costs in most systems. This is avoidable in sender-initiated strategies by ensuring that load balancing is performed only when new tasks are spawned. This advantage may however be lost in systems executing tasks of unequal lengths where preemptions and migration of executing tasks are required to ensure that all processors are equitably loaded.

The primary focus of this paper is to explore a new strategy for dynanic load balancing in hypercubes. We describe a user transparent, distributed, two-tiered graph_theoretic algorithm to dynamically allocate tasks onto the different nodes. A receiver-initiated strategy is adopted wherein the underloaded processors broadcast there status to the neighbouring nodes enabling the saturated processors to construct domains within which they could redistribute the load. These " balancing domains " or " pebbles " so constructed are represented using attributed hypergraph data structure. These domains are not however immediately closed because at the time of process migration tasks may be dispatched to processes embedded in other pebbles. So in order to actuate the distribution the pebbles are transmitted to their respective cluster controllers which compute an optimal mapping of migratable tasks onto the underloaded processors. This is achieved through " pebble crunching ", which involves controlled fragmentation and recombinations of pebbles owned by different nodes. This modified schema is then redistributed among the candidate nodes to actually carry out the load partitioning. The algorithmic details are described in the subsequent sections.

2. PRELIMINARIES

2.1 ENVIRONMENT CHARACTERISTICS

This load balancing schema is primarily targeted loosely coupled, *computation ensembles* with n homogeneous processing elements interconnected through a broadcast based communication subnet instead of shared variables. The interconnection network topology may be of the type of two-dimensional, spanning bus hypercube, toroid, 2-ary N-cube, hypertree or cube-connected cycles. A common characteristic shared by these interconnection networks is the high degree of interconnectivity. In addition the following characterize the architectural properties of the proposed model.

[1] the processing nodes in the ensemble are *homogeneous*, in that a job submitted at any node in the network may be processed at any other node in the network

[2] the node behaviour is heterogeneous in that tasks are spawned, destroyed or arrive from the external hosts, at arbitrary rates on the different nodes. Placement of new external tasks on the processing nodes is either done by the user or by a host processor whose primary function in most systems is to serve as an input/output device. Consequently response time is different for each node depending upon the computational requirements of the tasks and local availability of resources and precedence-constraints among the tasks.

[3] there are reliable, error-free, full duplex communication links between the processing nodes. The network communication protocol is completely separated from the inter-task communication policy.

[4] there is no intermediate buffering of data and control messages. The messages are received by a node from a remote node in the order in which they are transmitted. This is a difficult assumption to satisfy in loosely coupled homogeneous ensembles as a significant number of messages reach out of order due to channel contention and process priorities. However if the operating system implements virtual time, as is the case for some of the recent versions, then this assumption can be met.

[5] an executing process is interrupted by any control messages directed to the node on which it is executing and the nodes have the ability to distinguish between different types of messages when operating in the asynchronous mode.

2.2 Problem Descriptors

Load balancing calls for an optimal task distribution in a configuration space with conflicting demands [1]. In order to avoid *processor thrashing* or excessive accumulation of load on any processing node and to achieve maximal utilization of system resources the tasks need to be spread out evenly over all the nodes. On the other hand the goal of minimizing interprocessor communication to prevent channel saturation requires that tasks be clustered on few, adjacent processor-nodes. This necessitates a two-tiered solution to the problem alongwith a classification of constraints into two broad categories, *processor-workload characteristics* and *process-interaction characteristics*. The former serves as a thresholding parameter to initiate load balancing while the latter are a function of processor utilization, queue length, memory requirements, task mix, resource requirements etc. The latter are used to decide on how to actually distribute the load and refer to the process management overhead and the degree of reduced network usage as a consequence of process-migration, breakage and reestablishment of inter-process communication links, precedence constraints etc.

2.2.1 Processor_Workload Characteristics

The load of each processing node P_i is determined by the number of tasks currently being served, blocked i.e hanging at synchronization points or queued at that site. For the ith processing node we define a *threshold load* which is defined to be the loading condition for a processor such that further addition of tasks to it leads to no further gain in processor utilization. Based on the instantaneous task load we quantitatively define the loading states for a processing node to be *excessive, optimal* and *light*. An excessively loaded node can get rid of some of its present load while a lightly loaded node could absorb more load. If the system is in neither of these states then the loading is optimal. As stated earlier the system tends to improve throughput by avoiding idle or lightly loaded processors. We also define the notion of *Balancing Region*, BR_i, for a processor i which includes all prospective candidates for receiving tasks. This region can be defined statically and changed dynamically depending upon the state of the system. So task migration is essentially a comparison between the degree to which the load distribution of the balancing region is unbalanced and the loading threshold. To quantitatively measure the degree of "balancedness" of a system, Livny's [9] Unbalance Factor defined over a balancing region may be used, which is given below

$$UBF(i,t) \stackrel{\Delta}{=} \begin{cases} \infty & (\hat{\Delta}L(A,t) > 1) \wedge (min_{j \in BR_i(t)}(n_{i,j}(t))=0) \\ \frac{\hat{\Delta}L(A,t)}{\min_{j \in BR_i(t)}(n_{i,j}(t))} & (\hat{\Delta}L(A,t) > 1) \wedge (\min_{j \in BR_i(t)}(n_{i,j}(t))>0) \\ 0 & otherwise \end{cases}$$

where $\hat{\Delta}L(i,t) \stackrel{\Delta}{=} \max_{k \in BR_i(t)}(m_{i,i}(t) - m_{i,k}(t))$ is the relative load-difference of i at time t. $BR_i(t)$ refers to the balancing region for a processor i at time t, which here refers to the hypercube dimension D, and $m_{i,j}(t)$ denote the number of tasks at processor j. So then, given the load vector specifying the instantaneous load at each node determine an assignment such that unbalance in the system and total communication costs, measured as the sum of total data transfers between the nodes are minimized.

2.2.2 Processor_Interaction Characteristics

The availability of a neighbouring underutilized node alone does not merit load sharing, specially if the process migration overhead and interprocess communication link breakage and reestablishment were to lead to a greater turnaround time for the migrated task than if it were to be locally processed. So a quantification for the message passing overhead due to precedence constraints or synchronization requirements and the parameters affecting process-migration is needed to ascertain the effectiveness of balancing alternatives.

Computations intended to run on concurrent computational ensembles are decomposed into set of tasks which could then be concurrently executed. This problem decomposition often induces precedence constraints among the tasks which the distributed nature of the computational system translates into message passing requirements. These message passing requirements due to precedence constraints are determinable at load time when the tasks are pumped to the various nodes by the host processor or at process creation time if they are dynamically spawned. Each task is provided with a list specifying the messages it sends to the other nodes with their addresses, list of node addresses from which it is to receive messages and the length of messages. Thus the task coupling function due to precedence constraints is specified apriori and is used here as a decision criterion. Let $\Pi^k{}_{ij}(\delta_i \delta_j)$ denote the length of k th message exchanged between the processes executing on nodes i and j. If the channel capacity in bytes/sec is denoted by D, the total message passing overhead for a process executing on node i that needs to be incurred before it is completed is given by,

$$M_{pass_i} = \sum_{j \neq i} \sum_k [\frac{\Pi^k{}_{ij}(\delta_i \delta_j)}{D} + Z_{ij}\ F\ (\Pi^k(\delta_i \delta_j))$$

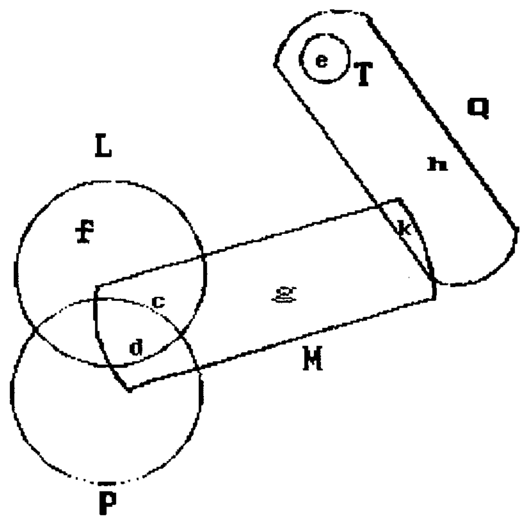
where Z_{ij} clock cycles is the fixed protocol and routing overhead and is in general a function of the node on which a task is executing. In addition, the overhead for migrating the task and its state tables, communication link breakage and reestablishment need to be accounted for. So the total interprocess communication cost for migrating a task, T, from processor i to j is denoted by

$$MIG_T = M_{pass_T} + TSIZE_{ijT} + \sum_j \Phi_{ij} + TAB_{T_{ij}}$$

where $TSIZE_{ijT}$ denotes the propagation delay for migrating task T, from node i to node j. TAB_T refers to the cost of migrating all state tables and process control block, (PCB) pertaining to the task and Φ_{ij} denotes the overhead of breaking and reestablishing all links for task T executing on node i. The latter pertains to the message passing overhead involved in transmitting control packets to all the nodes communicating with the task on node i.

2.3 Constraint Representation Using Hypergraphs

Using the Unbalance factor the feasible balancing horizons for the heavily loaded nodes are constructed if they exist i.e there exist neighbouring nodes which are lightly loaded and can accept excess load. However these domains are not closed as they are constructed, in that they are candidates for furthur optimization. This is desirable from several standpoints. For example, as shown in Fig. 1, nodes c and f are in the balancing horizons of L, M, and P. Now, if all of them were to send there excess tasks to C it would immediately get saturated and load balancing would again be required, incurring a heavy overhead in repeated process migrations. Also a selection between the nodes c and f needs to be made since at this stage they are both contenders for sharing load with nodes L, M and P.



This necessitates incorporating a global decision mechanism, which using the process-interaction characteristics determines a mapping of migratable tasks onto the appropriate processing nodes such that processor utilization is enhanced. But to perform such an optimization, the process-interaction characteristics and relevant system information need to be communicated to a controller node, which can then make the balancing decisions. This objective entails a data structure which can effectively bridge the gap between representation domain used for load balancing and the information required to compute a distribution mapping. To this end we propose the usage of *attributed hypergraphs* to express the optimization constraints and encapsulate the dynamic structure of balancing domains. Some of the terminology pertinent to the model is presented below.

Definition 1: Let $X = \{x_1, x_2, .., x_n\}$ be a finite set, and let $E = (E_i \mid i \in I)$ be a familiy of subsets of X. The family E is said to be a *hypergraph* on X if

(1) $E_i \neq \Phi \quad (i \in I)$

(2) $\bigcup_{i \in I} E_i = X.$

The couple $H = (X, E)$ denotes a hypergraph. The elements of X are called vertices and elements of E are called hyperedges.

Definition 2: An *attributed hypergraph* is one whose vertices are associated with a nominal list of numerical attribute values. Here it is used to model the loading state of the system where each attributed hyperedge corresponds to the domain of processors over which a heavily loaded node may distribute its excess load and the vertices in the hyperedge represent the underutilized processors.

We further introduce the notion of a **Pebble,** which is the fundamental unit of information interchange in the model. Pebble is an attributed hyperedge associated with some owner node,P , which is not itself a component of any other pebble. It is denoted by $E_P = [(x_i, k, [A]) \mid x_i \in X$ and $k \in N$ and $[A]$ is set of m-attribute tuple denoting the cost of migrating m-excess tasks. So a pebble has the following structure, $(x_i, k, (a_{11}, a_{12}, .., a_{1k})), (x_j, (a_{21}, a_{22}, .., a_{2j})), ..., (x_m, l, a_{m1}, a_{m2}, .., a_{ml}))]$, where $x_1, x_2, .., x_k$ are the lightly nodes with which the *pebble-owner* could share its excess taskload; k,j and denote the accepting capability of receivers. $(a_{i1}, a_{i2}, .., a_{ik})$ is the attribute tuple where a_{ij} denotes the cost of migrating the jth excess-task to node i. A pebble is also referred to as *Local Balancing Horizon* of an overloaded node, or LBH.

Definition 3: A *Pebble Cluster* corresponds to the attributed hypergraph constructed by the Distribution Cluster Controller upon receiving the pebbles from all the heavily loaded nodes in the cluster. Fig. 1 illustrates a pebble cluster. Initially each cluster has several overlapping pebbles with overlapping vertices denoting the contending receiver nodes which can receive tasks from more than one overloaded node in the cluster. Further the ensemble itself, may consist of overlapping pebble clusters.

3. STATIC NODE CLUSTER FORMATION

In order to enforce a hierarchial control the nodes in the ensemble are partitioned into static clusters and DCCs are elected for each cluster. The common criteria for clustering being minimization of internodal communication cost, connectedness or k-link failure resilience i.e the nodes remain *strongly connected* upto k link failures, minimization of routing tables or a balanced structure with respect to certain metrics e.g computational power, size etc. As the processors and communication links have been assumed to be reliable and stationary, the criterion adopted here for clustering is balance i.e the processing nodes are partitioned into clusters of approximately equal size for the purpose of load balancing where the clusters are in the form of route balanced m-ary trees. Given the degree of each node in the m-ary tree and the number of clustering levels desired the clusters can be constructed using the bottom-up algorithms proposed by by Ramamoorthy et al, [14]. The root of each m-ary subtree at each level is designated as the DCC. The parent of each DCC then becomes the cluster controller for the next level of hierarchy. This process is repeated upwards till the root of the tree which is designated as the System Cluster Controller. All the controllers above the level of DCC are elected from among the DCC nodes to save on communication overhead in transmitting the partially computed allocations. Each node is then made aware of its own controller node. Fig. 2 shows a 16 node hypercube partitioned into hierarchical static clusters.

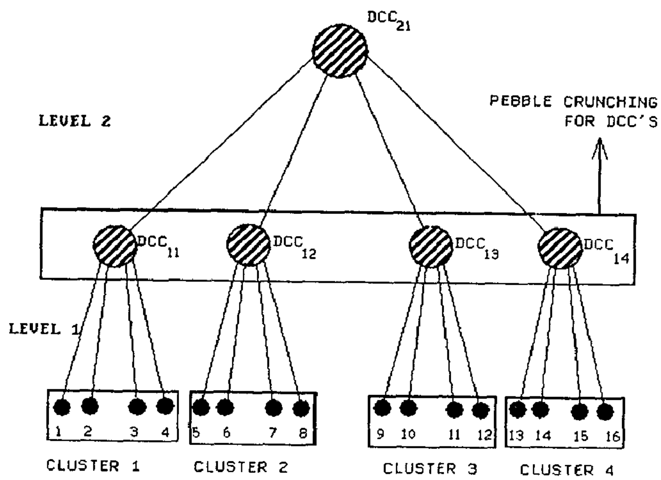


FIGURE 2. CLUSTER TREE FOR A 16 NODE ENSEMBLE

Since clustering is used primarily for the purpose of reducing communication overhead for control, the nodes in one cluster are not forbidden from sharing their excess tasks with nodes in another cluster. In fact the LBH of overloaded boundary nodes will contain nodes in the adjacent cluster. However the decision to share tasks with nodes in other clusters are taken after pebble crunching is completed within a local cluster and the crunched pebbles are communicated to the next level controller. If a particular DCC is the next level controller then it is required to send request packets to all lower level controllers asking for pebble clusters. This process is recursively folded upto the root of the cluster tree.

4. LOAD BALANCING ALGORITHM

Based on the above preliminaries we now present the load balancing protocol involving four phases. In the first phase each processor determines its loading state. If a processor is not being fully utilized due to process or data unavailability, it conveys this information to all its neighbouring nodes. In the next phase, all the overloaded processors in the neighbourhood of the lightly loaded node, use this information to construct their local balancing horizon, which is represented by a hyperedge of an attributed hypergraph, also denoted as a *pebble*. In phase 3, this information is transmitted to the *Distribution Cluster Controller* (DCC) for that processor which constructs a pebble cluster from the pebbles. The complete hypergraph or pebble cluster, encompasses all the feasible reassignments for that cluster. This pebble cluster may however contain several overlapping pebbles i.e two or more excessivley loaded nodes which can share their load with the same set of underloaded nodes. Transversals are then computed for this hypergraph and the overlapping nodes are assigned to one of the nodes containing them in their local balancing horizon. The crunched pebbles are rebroadcast to their respective owners which can then distribute the load in the new balancing regions.

PHASE I: RECEIVER-INITIATED LOAD REQUESTS DURING THRESHOLD DEPLETION

Before an instantaneous global scenario can be constructed for generating the load distribution, its constituent components are composed using the information broadcast by the processors regarding their loading conditions. The imbalance function is used by each processor to determine if it can benefit by accepting or by ridding itself of additional load. If there is a processor which can accept additional tasks for execution i.e it is a underloaded node, then it broadcasts this information alongwith the excess capacity to all its neighbouring processors. This state recording and broadcasting algorithm is superimposed on the underlying computation. Since all state communication is interrupt based, the arrival of a *control packet* from the adjacent lightly loaded node, forces the destination processor to interrupt processing, and examine the incoming message. If the interrupted node is also operating below its threshold capacity then it ignores this incoming information and continues processing. An excessively loaded node however extracts out the address of the sending node. The information pertaining to all the neighbouring processors with which it could possibly share its load, is collected to construct or update the *local balancing horizon* (LBH). The Local Balancing Horizon of a heavily loaded processor then is a domain of underutilized, neighbouring processors over which it could distribute its excess tasks. There are two observations regarding the the construction of LBH as given below.

OBSERVATION I. As there is no global system clock controlling these events and each processor records and transmits its state independently, a mechanism is needed which will enable a heavily loaded processor to know when all the lightly loaded nodes have communicated their status as their number is not known a priori. So the heavily loaded processor could be made to wait for $\sum_{i=1}^{d} T_{ij} + \phi_j$, from the time of arrival of the first control frame, where T_{ij} is the propagation delay for a control frame from node i to reach node j. The buffering and protocol overhead is denoted by ϕ_j and d refers to the fanout of processor j. At the expiry of this interval the pebble construction is initiated and any packet which arrives late is rejected, for the current balancing cycle. This ensures that nodes do not wait infinitely for the prospective receivers. However, most of the targetted hypercube computational ensembles are loosely synchronized and cannot be expected to display preset, collective behaviour.

OBSERVATION II. Rather than providing each node with a deterministic, repetitive control, the LBH construction algorithm is made completely asynchronous and distributed. Instead of transmitting loading states at regular intervals, the state changes are broadcast as they occur, i.e the LBH is continuously updated and monitored. Each time there is a state change in some processing node, for example a heavily loaded node becomes a lightly loaded node, it dissolves its own LBH and broadcasts this state change to its neighbours. On the other hand if an underloaded node receives tasks for execution and exceeds the optimum loading level, it transmits a control frame to this effect so that all overloaded nodes in its immediate neighbouhood can delete it from their LBHs. With this strategy each update in the LBH requires broadcasting one control packet. This approach does not impose additional synchronization overhead as the overloaded nodes do not need to wait for all underloaded nodes in the neighbourhood to communicate their status before it can construct the pebble to send it to the DCC. On the other hand, the LBH is dynamically updated and can be transmitted immediately upon request to the DCC.

PHASE 2: PEBBLE CONSTRUCTION AND ATTRIBUTE ENCAPSULATION

In this phase, the heavily loaded sender nodes construct pebbles to transmit their local balancing horizons to the DCC upon request. On the basis of their current loading state and the threshold each overloaded node determines its *shareable task set*, which contains a list of excess tasks that may be migrated to other nodes. For each task, T, in the shareable task set it computes the balancing delay and the protocol overhead, MIG_T and uses it to construct its pebble. This pebble denoted by E_i, is then transmitted to the Distribution Cluster Controller for crunching, i.e to determine the globally optimal load distribution

PHASE 3: TRANSVERSAL COMPUTATION AND GLOBAL BALANCING

During this phase all pebbles in a cluster are centrally operated upon to determine task reallocation schema. The DCC collects all the pebbles dispatched by the nodes for which a LBH exists. If a heavily loaded node does not have any underutilized processor in its neighbourhood then its LBH is empty. A node with an empty LBH may be required to share its load with processors more than one hop away. Thus the LBH of such a processor includes all the underloaded nodes in the nodal cluster. All such nodes send a control message to the DCC, informing it of the unavailability of local nodes for sharing their load. The DCC then constructs the pebble for them. In this phase the loading state of the entire system or the pebble cluster is modeled as an attributed hypergraph where each pebble denotes a hyperedge. A hypergraph is said to have a *cycle* if there is a hyperedge that is a subset of some other hyperedge. The immediate implication of the existence of a cycle in the hypergraph being that there is a pebble whose owner can share tasks with a node or a set of nodes which is a subset of nodes with which another pebble could share its tasks. It denotes a maximal sharing conflict situation. The load sharing algorithm involves determining all such cycles at the outset and reducing them by allowing their owners to share their excess tasks with the nodes in the cycle. After reducing all proper cycles in the pebble cluster we need to determine the *minimum transversals* with respect to each pebble to find out all possible conflicting load assignments or those nodes which are in the shareable task set of two or more nodes.

Definition 3: A *transversal* of a hypergraph H = (X; $E_1, E_2, E_3, \ldots E_m$) is defined to be a set $T \subset X$ such that

$T \cap E_i \neq \phi$ (i = 1,2,3,...,m).

where the minimum transversal is defined by the set $T \cap E_i$.

Definition 4 : The *transversal number*, $\tau(H)$ of a hypergraph is defined to be the minimum number of vertices in a transversal, and is denoted by $\tau(H) = \min|T|$.

In this method the minimum transversal with respect to each pebble, in the attributed hypergraph is needed to determine the globally balanced task allocation, for all the excessively loaded nodes.

Berge [2] has described an algorithm for determining the minimum transversal Tr A which is summarized below

STEP 1. Determine the set of all minimal subsets of A i.e Min A = { $A_1, A_2, \ldots, A_k$ }.

STEP 2. successively determine the following families:

$A_1 = A_1 \rightarrow \text{Tr } A_1 = (a \mid a \in A_1)$

$A_2 = A_1 \cup A_2 \rightarrow \text{Tr} A_2 = \text{Min}(\text{Tr} A_1 \vee \text{Tr } A_2)$

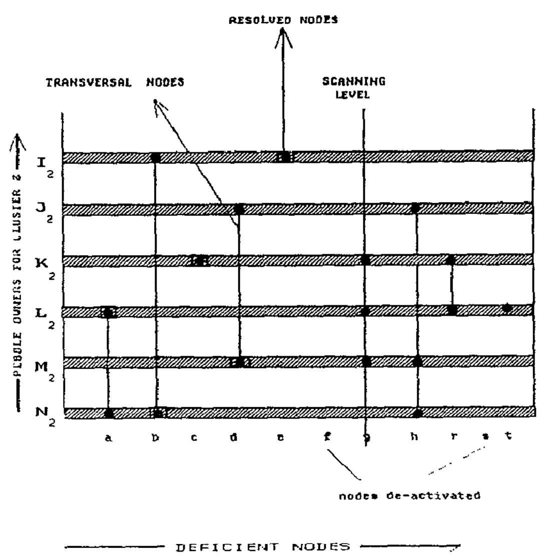
$A_3 = A_2 \cup A_3 \rightarrow \text{Tr} A_3 = \text{Min}(\text{Tr} A_2 \vee \text{Tr} A_3)$

etc.

Using $Tr\ (A \cap B) = Min\ (Tr\ A \vee Tr\ B)$, the $Tr A_{k+1}$ can be computed from $Tr A_k$. If there are k excessively loaded nodes in the system whose LBRs have been submitted to the DCC then this algorithm constructs Tr A = Tr A_k in k steps.

This algorithm computes the composite minimal transversal sets for the entire hypergraph, i.e it determines all the conflicting nodes in the system. But it cannot be used per se because with each node in the conflict set there is no information regarding the owners. The above algorithm could however be modified to compute the minimum transversal with respect to each pebble owner, such that as soon as a node is detected which can share tasks with two nodes a task sharing decision is taken. But this underutilized node may be in the pebble of some other node with a lower balancing cost, in which case a nonoptimal assignment would have taken place. So a mechanism is needed to determine all pebble owners with respect to each conflicting node in the pebble cluster. Using the Process_Interaction characteristics we could then decide upon the minimal cost task migration among the various candidates.

Since the ensemble nodes have been assumed to be reliable and stationary, and a static clustering algorithm is used to partition them into clusters and the cluster controllers are aware of the nodes in their respective cluster, we can create a data structure at the cluster controller which considerably simplifies minimum transversal computation. At each cluster controller a k **X** k+l, boolean bit matrix which is maintained as a boolean template. It is constructed using the information contained in the pebbles where k corresponds to number of nodes in the cluster and l denotes the number of overlapping boundary nodes belonging to a different cluster which could be in the LBH of two nodes. Also two pebbles cannot have more than one boundary node in their LBH thus placing a bound on l. An element b_{ij} of this boolean matrix is set to 1 if node j is in the LBH of i. The minimum transversal computation then reduces to filling the template and scanning along the columns for more then one 1. So K+lth minimum transversal can be computed in $O(k^2)$. Fig. 3 shows how the template is used to compute minimum transversals with respect to each pebble in a given cluster. For details refer [11]. The nodes in a minimal transversal are considered equivalent i.e a processor may distribute its load to either of these nodes or divide the excess load equally among all the nodes in the minimal transversal. Pseudo-code for the algorithm to determine global balancing schema is given below.



Boolean Pebble Crunching Template

```
algorithm PEBBLE_CRUNCHING (A,E);
/* A is the set of processors in the system and
   E denotes the pebbles or attributed-hyperedges */
   begin
1.    partition E into sets C and E-C
      using Graham Reduction;
                  /*C is the set of all proper
                    cyclic-hyperedges in the
                    hypergraph */
2.    assign and mark processors in each set in C to
      their pebble-owners and dissolve cyclic pebbles;
3.    for all empty pebbles do
4.       E_j = {E-C};
               /* balancing domain of empty
               pebbles is the entire system */
5.    for all pebbles in E-C do
        begin
6.        compute_transversals(E-C);
7.        assign nodes in TR(E-C);
               /* assign nonconflicting nodes */
8.        for all nodes in TR(E-C) do
            begin
9.            form sets, OWNER_i of pebble
              owners for each node in TR(E-C)
10.           for all sets OWNER_i do
               begin
11.               using the attribute tuple extract out
                  all tasks that can be assigned to
                  node i;
12.               perform the minimal-overhead
                  task assignment among the several
                  owners;
13.               mark nodes in TR(C) assigned;
              end
          end
      end
14.   for all pebbles in {E} do
15.     if a task a_ji in attribute-tuple
        is assigned to a
        receiver in same tuple then
16.        replace all (x_i,k,{A})
           tuples with (x_i,a_ji);
              /* the attribute list in
                 hyperedge is
                 replaced with the
                 processor-task pairs */
   end;
end        /* pebble_crunching */
```

Each DCC runs the algorithm *Pebble_Crunching* to construct a task redistribution schema which is optimal within a cluster. These crunched pebble clusters are then passed onto the cluster controllers at the next level in the hierarchy. As mentioned previously the higher level controllers are elected from among the DCC's to reduce the number of nodes involved in control functions and to reduce on communication costs. This computation is recursively carried upto the root of the cluster tree, which then returns the crunched pebbles to their respective owners.

PHASE 4: PROCESS MIGRATION

When the SCC completes pebble crunching at the root of the cluster tree, it needs to return the pebbles to their respective owners. The returned pebbles have the form $E_i = [(x_1,a_1),(x_2,a_2),..,(x_j,a_j)]$ where x_j denotes the node j to which the excess task with balancing cost a_j is to be transferred. These pebbles now contain the closed balancing horizons which are globally optimized. On receiving the distribution schema the owners i.e the heavily loaded nodes transmit their excess tasks to the nodes in the pebble. The migration phase involves detaching processes from their current environment and transfering them to new nodes and then reinstalling them. The processor dependent locations need to be remapped onto the new node, alongwith the redirection communication links between the processes. The process migration mechanism also needs to ensure that all processes with which this process requires to communicate are informed of its changed location. While this process is still being migrated the process migration mechanism needs to buffer all the incoming messages for this process and pass them on to it, after its installation is over, because when the process is migrating others continue to interact with it.

5. SIMULATION FOR PERFORMANCE EVALUATION IN LARGE SCALE SYSTEMS

A number of simulations were carried out on the VAX 11/780 to evaluate algorithm performance to analyse the gain and variations in throughput, execution speedup and processor utilization with load balancing as the systems are scaled up. The simulation involved randomly generating large task sets and their computational requirements, precedence graphs, nodes on which they were to be externally submitted or dynamically spawned, creation times, and interprocess communication requirements. Data used during the simulation was obtained from benchmarking tests conducted on the NCUBE hypercube and results were averaged over several runs.

Figure 6.1 shows the variation in processor utilization with the number of nodes, with and without load balancing where processor utilzation is defined as the percentage of time the processing node is busy. The processor utilization was averaged over all nodes in the systeem. The graph shows that for a constant number of tasks, as the system was scaled up there was a decrease in processor utilization which could be attributed to increased delays during process synchronization and process migration. Fig. 6.2 demonstrates the variation in the ratio of useful machine cycles to the total machine cycles performed by the processing nodes versus the number of nodes in the system. The fall in the ratio as illustrated by the results is to be expected because as the number of nodes is scaled up the load balancing overhead increases as the number of levels at which pebble crunching is performed increases. Useful cycles here refer to the computational cycles spent on task execution and excludes the overhead for load balancing, process migration and context switching.

Fig. 6.3 illustrates the increase in throughput as the number of nodes is increased. Though increase in throughput is implicit here due to increase in concurrency it is seen that pebble crunching load balancing algorithm leads to enhanced throughput than in

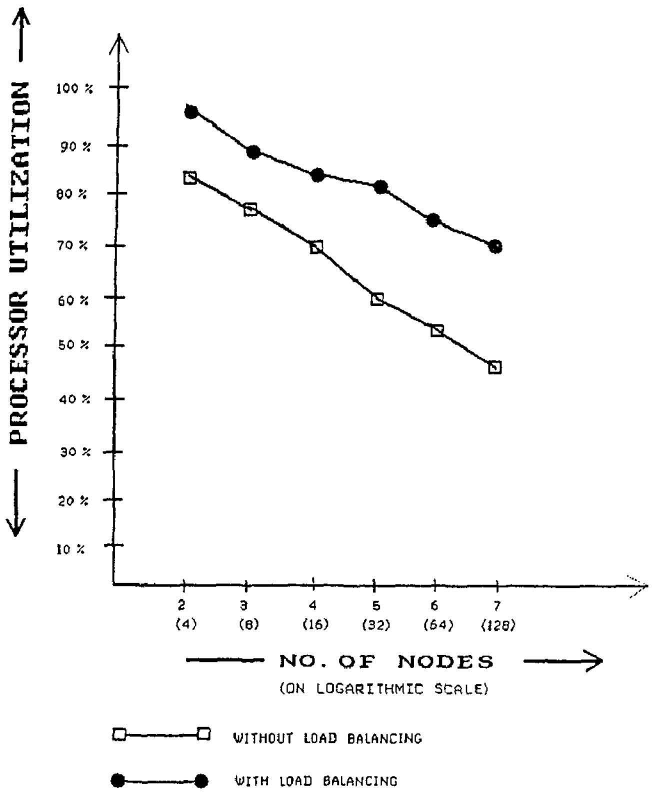


Figure 6.1 : Processor Utilization Vs No. of nodes

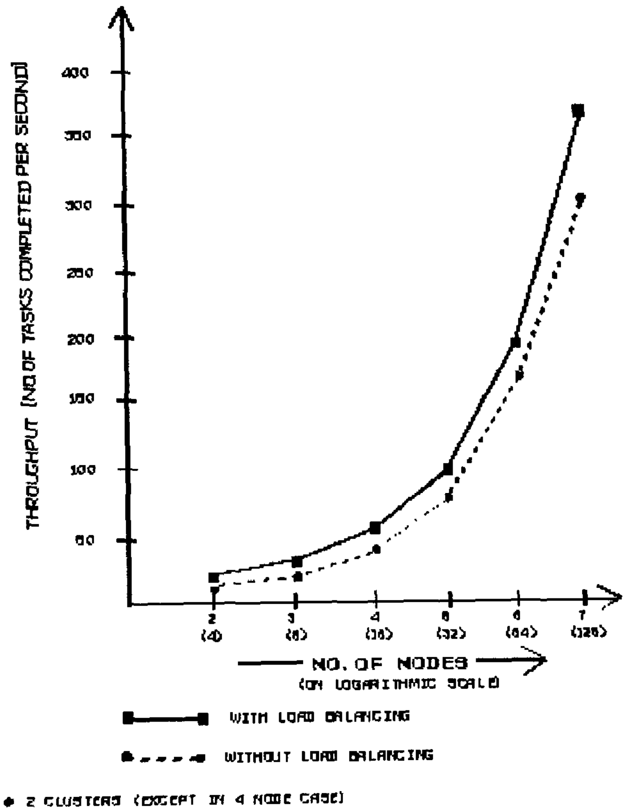


Figure 6.3 : Throughput VS No. of nodes

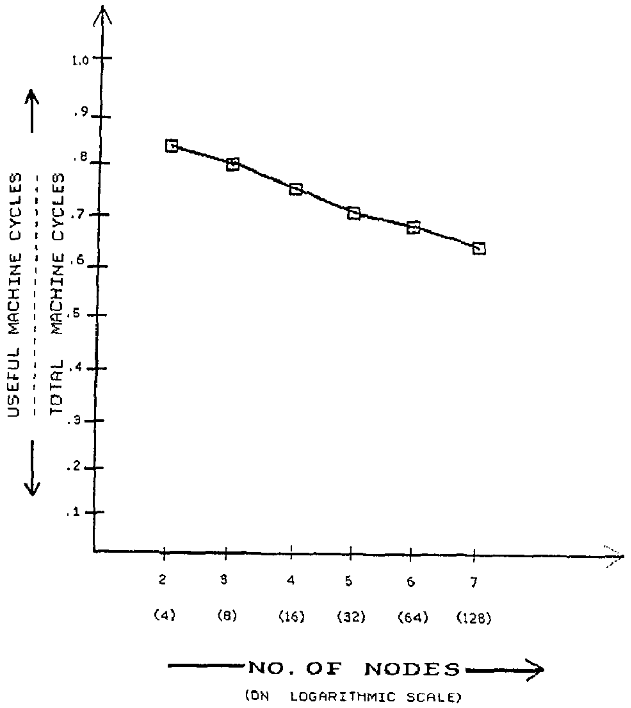


Figure 6.2 : Variation of useful work with the No. of hypercube nodes

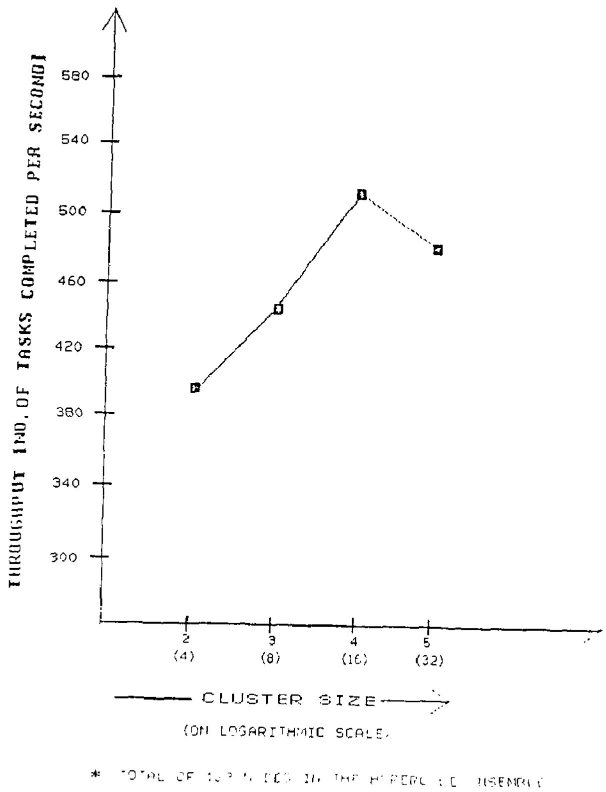


Figure 6.4 : Throughput VS Cluster Size

systems with no load balancing. What is more significant from the graph is the fact that gains over systems with no load balancing are higher as the systems are scaled up.

The computational overhead incurred by the algorithm is heavily dependent upon the cluster size as addition of clusters leads to an increase in the number of pebble crunching cycles. Fig. 6.4 analyses this relationship between the cluster size and throughput. It was seen that throughput does not monotonically vary with cluster size. For large scale systems, with smaller cluster size the cluster tree is large and pebble clusters have to transmitted several times before a load distribution is arrived at. With a large cluster size this process is reduced. However in the latter case the message passing overhead is higher as more local balancing horizons need to be transmitted and pebble crunching takes place over a larger set.

6. INCORPORATING FAULT TOLERANCE

The nature of applications envisaged for the hypercube ensembles impose high demands on reliability, availability and performance. Consequently the traditional approaches for achieving fault tolerance through modular redundancy and circuit duplication or voting are not adequate as the number of standby redundant spares need to be kept to a minimum for extracting maximal concurrency from the system. So mechanisms are needed for concurrent fault detection and fault isolation. We now drop the assumption in sec. 2.1 and consider both nodal and link failures. It is assumed that the Distribution Cluster Controllers will occassionally fail and new DCCs will need to be elected and the nodes in the cluster made aware of their presence. This may lead to a redefinition of the nodal cluster itself and subsequent invalidation of the transversal computing boolean template. Here we do not concern ourselves with the implications of the failure on the ensuing computation at a node but instead on its implications on the load balancing process. For example, if a DCC were to fail then there would be no load balancing within a cluster. The pebbles would be transmitted to it but they will not be crunched and returned.

In the past hardware solutions have been proposed for providing fault tolerance in the event of node/link failures (Rennels [15]). Rennels's method attempts to maintain original performance and connectivity by adding another dimension to the hypercube to provide spares. Each node in the basic hypercube is provided with an additional serial port which is used to reach a spare node. A group of nodes is connected to the same spare via a crossbar matrix such that if any of them fails the additional node can be inducted into the system such that all neighbouring relations are preserved. Our model can be implemented per se under this framework as this processor shuffle does not disturb the topology of the ensemble and static cluster organization is valid. However this implementation is available in very few prototype systems. So we propose a distributed algorithm for detecting and recovering from failures which does not require any hardware modifications.

Our strategy is based on the PMC model introduced by Preparata et al [13]. Its essential characteristic is to decompose the ensemble into subunits which are capable of testing each other. By this method, a fault-free unit will ultimately detect a faulty node which can then be excluded from the processing set. If the faulty node turns out to be a DCC then another DCC needs to be elected and its existence made known to the others. Through simulations it was found (section 6) that there is an optimal cluster size for a given hypercube dimension at which the pebble crunching yields maximal throughput. So initially the ensemble is divided into optimal size clusters. As the computations proceed in time some processors or links may fail, varying the effective cluster size. The fault diagnosis algorithm is implemented within each cluster to maintain a conceptual consistency with the load balancing algorithm, though it need not be so. Each cluster is denoted by a configuration graph [11], K = (N,L) where N is the set of processors and L is the set of physical communication channels. Also each node and edge is associated with an "active" or "failed" state denoted by a boolean variable. These individual states are combined to form a *diagnostic state vector* which epitomizes the active state of the entire configuration graph, denoted by $S = \{ nb_1, nb_2, ..., nb_N, lb_{1,2}, lb_{2,3}, ..., lb_{N-1,N} \}$. Further *diagnosability*, of nodes and links from the perspective of an intact node, has been defined [11] as,

Definition A node n_j is diagnosable from n_i if there exists a path in the configuration graph

$$P_{i,j} = < n_i, l_{i,k}, n_k, l_{k,r}, ..., n_j, l_{m,j}$$

where n_i , n_k etc. are operational nodes. A communication link $l_{j,k}$ is diagnosable from node i if nodes n_j and n_k are diagnosable from n_i. In a hypercube of dimension k, each node can diagnose upto k nodes and broadcast their status to the others nodes in the cluster. This requires that the communication subsystem does not fail, in that diagnostic messages can be sent and acknowledged within fixed time intervals if the node being checked is functional. Since each node has an independent communications processor, node failure can be distinguished from link failure. But a confirmed node failure results in all its outgoing links being declared faulty too.

For our purpose we do not want tasks to be transmitted to failed nodes or over non-operational links. Since this model is receiver_initiated a node will be able to convey request for additional workload only if it is intact and receipt of this request implicitly implies link availability. But it may fail after the load-request has been broadcast to the neighbours and any processes migrated to it after crunching will be lost. To prevent this task loss and reduce the message passing overhead needed for diagnosing the health of receiver nodes the diagnosis defined previously is carried out after the crunched pebbles have been returned to their owners i.e before process migration and tasks transmitted to a designated receiver only if it is determined to be intact.

In contrast to the traditional approach to fault diagnosis whereby all nodes in the system are uniformly diagnosed we introduce a distinction in our model. Only the functionality of elected DCCs is diagnosed by their immediate neighbours at regular intervals. Results of this neighbourhood test are then broadcast to other nodes in the cluster. Links are also validated by this process with each node updating its diagnosibility state vector, S, for the path to the current DCC. As the inter-nodal communication paths in the hypercubes are directly determined by the addresses of the source and destination nodes and their hamming distance we do not need to maintain the complete diagnosibility vector and only the address of current DCC and knowledge of failed neighbouring link is adequate. If a DCC failure is detected then another will have to be elected in accordance with some protocol that ensures a unique winner.

There are two basic approaches to "elections in a distributed system" [6], *bully approach* and *invitational approach* which can be used to select unique controllers from among a collection of nodes. In the former, a priority schema is used to order the processors and the one with the highest priority from among the nodes participating in the election will become the new DCC. In the invitational approach the nodes which wish to become controllers invite other nodes to join them as a group. As mentioned earlier this model carries out recovery within the clusters setup initially for performance optimality. So we use the bully election approach to re-elect DCCs in which the nodal address serves as a priority metric. In accordance with the binomial spanning tree, (BST), paradigm which encaptures the addressing and connectivities among the hypercube nodes we designate a higher priority for nodes whose addresses have lower boolean value.

Pseudocode for algorithm for diagnosing the status of the current_DCC and reelecting a new one if it fails is given below. This is executed periodically on every node in the cluster. Since the size of a cluster is bounded from above, the boolean template used for computing the transversals during pebble crunching can retain its structure as shown in Fig 2. Every node is given a copy of this template which is activated when that node becomes the current_DCC. After a faulty DCC comes up it is allowed to become a controller again by participating in the elections after the active DCC crashes.

7. CONCLUSIONS

In this paper we have described a robust, demand-driven, distributed protocol for dynamic load balancing for heavily-loaded hypercube ensembles using a graph-theoretic approach. An attributed-hypergraph is used to model the instantaneous loading state for the system and provides a versatile mechanism for constraint representation and cost encapsulation. Unlike some of the existing algorithms this methodology ensures that process migration takes place only if it is profitable and with minimal overhead. Existence of lightly loaded processors is not seen as an adequate criterion for load sharing. This

```
procedure DCC_monitoring (i : node;
        current_clust_size : integer)

entry fault_monitoring ;
begin
  repeat
    delay( t : cycles );
    if i ∈ current DCC neighbourhood then
        send diagnosis_packet ;
        async_wait ( t2 : timeunits );
      /* wait for interrupt for upto t2 sec.*/
        if no response then
          broadcast DCC_Down ;
            set link (i,current_DCC) down ;
            update S_i ;
          /*S_i is the DSV for node i */
        else
            append l_{i,current_DCC},n_i to S_i ;
            broadcast DSV to neighbours;
        endif;
     else
        asyn_wait(t5 : timeunits );
      /* wait for DSV_pkt from neighbouring node */
       if DCC down then initiate elections for
                                 new DCC
       else update DSV and broadcast it ;
     endif;
   until forever ;
end /* procedure */

/* bully algorithm for electing a new DCC */

entry faulty_recovery ;
begin
  if current_DCC faulty then
      suspend load balancing procedures ;
      for all nodes in the cluster do
            determine if remote node k has address
          with lower binary value than i ;
          /* select only nodes with higher priority */
            send am_becoming_DCC packet to node k ;
      endfor ;
      asyn_wait( t3 : timeunits);
      if a node k with higher priority indicates
         a desire to become DCC       then
         i suspends its bid to be the new DCC ;
         asyn_wait ( t4 : timeunits );
       /* wait for id of new_DCC */
         set current_DCC to new_DCC ;
      else
         for all nodes in the cluster do
             determine remote nodes with addresses
           having higher binary value than i ;
           /* inform lower priority nodes /*
             send am_new_DCC packet to k ;
         endfor ;
      endif;
  endif
end   /* procedure */
```

Pseudocode for Fault Diagnosis and Reconfiguration algorithms approach is particularly desirable for large applications with spontaneous and erratic task generation at certain nodes leaving others underutilized. The algorithm described here dynamically recomputes the task distribution in a manner which enhances resource utilization by distributing load over underutilized processors. The algorithm avoids processor thrashing as it does not allow for arbitrary migration or broadcast of excess tasks to a lightly loaded processor beyond its excess capacity.

The model has also been extended to incorporate fault-tolerance in the event of node and link failures. It is ensured that tasks are not transmitted to nodes which have crashed or cannot be reached. On detection the faulty nodes are removed from the load balancing environment to prevent process loss. In the event of a DCC failure, contingency procedures are provided for re-electing another one from among the fault-free nodes in the cluster to take over the control functions. However to limit the amount of control traffic, fault diagnosis and recovery is performed at the cluster level rather than the system level.

ACKNOWLEDGEMENTS

This research was sponsored in part by Office of Basic Energy Sciences, U.S. Department of Energy under contract number DE-AC05-840R21400 with Martin Marietta Energy Systems through Oak Ridge National Lab.

8. REFERENCES

[1] BARHEN, J., " Combinatorial Optimization of the Computational Load Balance For a Hypercube Supercomputer ", pp. 71-80.

[2] BERGE, C., Graphs and Hypergraphs, North-Holland, 1973.

[3] CHOW, Y. and KOHLER, W.H., " Models for Dynamic Load Balancing in a Heterogeneous Multiprocessor Multiple Processor System ", *IEEE Transactions on Computers*, Vol. C-28, No. 5, May 1979, pp. 354-361.

[4] EAGER, D.L., LAZOWSKA, E.D. and ZOHARJAN, J., " A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing ", *Sigmetrics*, Vol. 13, No. 2, 1985, pp. 1-3.

[5] EFE, K., " Heuristic Models of Task Assignment Scheduling in Distributed Systems ", *Computer*, June 1982, pp. 50-56.

[6] GARCIA-MOLINA, H., "Elections in a Distributed Computing System", *IEEE Transactions on Computers*, Jan. 1982, Vol. C-31, No. 1, pp. 48-59.

[7] GULATI, S., IYENGAR S.S and BARHEN J., "The Pebble Crunching Model for Dynamic Load balancing in Homogeneous Computation Ensembles", (submitted for publication).

[8] LIN, F.C.H and KELLER, R.M, " The Gradient Model Load Balancing Method ", *IEEE Trans. on Software Engg.*, Vol. SE-13, No. 1, Jan. 1987, pp. 32-38.

[9] LIVNY, M., " The Study of Load Balancing Algorithms For Decentralized Distributed Processing Systems ", Ph.D Dissertation, Weizmann Institute of Science, Aug. 1983.

[10] LU, H. and CAREY, M.J., " Load-Balanced Task Allocation in Locally Distributed Computer Systems ", *Proc. 1986 International Conference on Parallel Processing*, 1986, pp. 1037-1039.

[11] MAEHLE, E., MORITZEN K., and WIRL K., " A Graph Model for Diagnosis and Reconfiguration and its Application to a Fault-Tolerant Multiprocessor System ", *IEEE Conf. on Fault Tolerant Computing*, 1986, pp. 292-297.

[12] NI, L.M., XU C. and GENDREAU, T.B., " Distributed Drafting Algorithm for Load Balancing ", *IEEE Trans. on Software Engg.*, Vol. SE-11, No. 10, Oct. 1985, pp. 1153-1161.

[13] PREPARATA, F.P, MERTZE G., and CHIEN R.T., "On the Connection Assignment Problem of Diagnosable Systems", *IEEE Transactions on Electronic Computers*, Vol. EC-16, June 1967, pp. 848-854.

[14] RAMAMOORTHY, C.V., SRIVASTAVA, J. and TSAI W., " Clustering Techniques for Large Distributed systems ", *Proc. 1986 Int'l Conference on Distributed Computing Systems*, 1986, pp. 395-404.

[15] RENNELS, D.A., " On implementing Fault-Tolerance in Binary Hypercubes ", *Proc. of the IEEE Conference on Fault Tolerant Computing*, 1986, pp. 344-349.

[16] STANKOVIC, J.A. and SIDHU, I.S., " An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups ", *Proc. of the 4th Int'l Conference on Distributed Computing Systems*, 1984, pp. 49-59.

[17] TANTAWI, A.N. and TOWSLEY, D., " Optimal Load Balancing in Distributed Computer Systems ", *Journal of ACM*, Vol. 32, No. 2, 1985, pp. 445.