Heuristic Process Migration for Dynamic Load Balancing in A Message-Passing Multicomputer[1]

Kai Hwang and Jian Xu

CENG Technical Report: 91-11

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562 (213)740-4470
kaihwang@panda.usc.edu

April 2, 1991

[1]This work was supported by an AT&T research grant to USC.

Heuristic Process Migration for Dynamic Load Balancing in A Message-Passing Multicompter[1]

Kai Hwang and Jian Xu

CENG Technical Report: 91-11

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562 (213)740-4470
kaihwang@panda.usc.edu

April 1, 1991

[1]This work was supported by an AT&T research grant to USC.

Heuristic Process Migration for Dynamic Load Balancing in A Message-Passing Multicomputer*

Kai Hwang and *Jian Xu*
Dept. of Computer Science
University of Southern California
Los Angeles, CA. 90089-0782

Abstract: In this paper, a new adaptive method is presented for dynamic load balancing on a message-passing multicomputer. The method is based on using easy-to-implement heuristics and variable threshold in migrating processes among the multicomputer nodes. This adaptive method uses a distributed control over all processor nodes as coordinated by a host processor. Based on this method, a distributed load balancer has been developed and implemented on a 32-node iPSC/386 multicomputer. Several benchmark programs were executed using this load balancer. Benchmark programs include recursive functions and backtracking search, which are often used in AI applications. Encouraging benchmarking results are herein reported, verifying the effectiveness of the new load balancing scheme. This adaptive load balancer is implementable on any multicomputer topology, although the reported benchmark results were generated from experiments on a 386-based hypercube machine.

Key Words: *Multicomputer, load balancing, adaptive heuristics, distributed operating system, process migration, recursive functions, backtracking search, and distributed AI processing.*

Contents

1. Introduction 1
2. An New Adaptive Method 4
3. Heuristic Process Migration 8
   A. Localized Round-Robin Method 8
   B. Global Round-Robin Method 9
   C. Localized Minimum Load Method 10
   D. Global Minimum Load Method 10
4.Operating System Support 11
   A. Process Control Block 11
   B. Operating System Directives 14
   C. Control Level Parallelism 18
5. Implementation of the Load Balancer 20
6. Experimental Benchmark Results 23
   A. Benchmark Programs 23
   B. Experimental Results 27
7. Conclusions 35
References 36

*This research was supported by an AT&T research grant to USC.

1 Introduction

A *multicomputer* is a multiprocessor system with distributed local memories. Each computer node in the system consists of a processor, a local memory and a switch connected to an interconnection network. The distributed local memories are not shared by the node processors. The communication among processor nodes is done via message passing. The entire ensemble is controlled by a *distributed operating system.* A multicomputer is used to involve all processor nodes working together in solving large scale problems. The purpose is to achieve time-critical execution of large-scale computations in either numeric or AI applications [10].

In using a multicomputer, a user program is often partitioned into subprograms and distributed to multiple processor nodes. At run time, each node starts with an initial process, which may create many child processes during the execution cycle of each subprogram. A processor continues to execute these child processes, until the entire subprogram gets executed. Even if starting with a fairly balanced initial allocation of subprograms, these initial processes may create vastly different numbers of child processes at run time. Figure 1 shows the unbalanced creation of processes in a 4-node multicomputer. To improve the resource utilization, processes created at local processor need to be remapped into remote processor nodes by means of migration. Such a dynamic load balancing is intended to keep equal amounts of workload on all nodes throughout the execution cycle [13].

The problem of dynamic load balancing has been studied by many researchers. Some methods use a queueing theoretic approach [5], [13], [19], where the system mean response time is to be minimized. Other methods use distributed control to balance the activities among multiple processor nodes [2], [17], [18]. In sender-initiated methods, the load balanc-

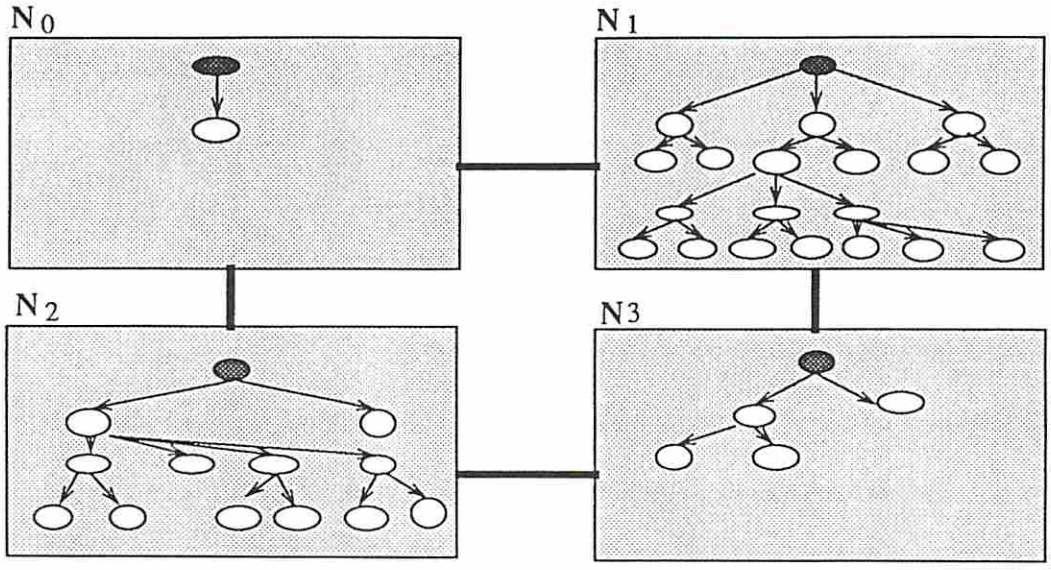


Figure 1: Unbalanced creation of processes in a multicomputer system.

ing activities are initiated by busier nodes [9] [11]; while in receiver-initiated methods, load balancing activities are initiated by lightly loaded nodes [12], [14]. A good dynamic load balancing method should reduce the overhead in collecting load indices and in process migration [6], [7]. The communication costs incurred in these operations depend on the message routing scheme used. In this paper, we present four easy-to-implement heuristic methods for dynamic process migration to achieve balanced load among multicomputer nodes. These heuristics avoid frequent load indices exchanges among nodes with the coordination of a host processor, such as the cube manager used in the iPSC system.

The term *threshold* has been used to decide when load balancing operations should be exercised [8]. Most load balancing methods use some fixed threshold values [3], [11], [14]. If the threshold is too low, the *thrashing* caused by excessive load balancing activities may degrade the performance of the entire system. If the threshold is too high, effective load balancing cannot be achieved at all. Recently, Pulidas *et al.* [15] proposed a gradient method to optimize the choice of threshold value. Their model requires frequent exchange

of load indices among all nodes without a centralized supervisor.

We propose a new scheme where each node updates the threshold on a periodic basis under the supervision of a host processor [21]. This idea is inspired by the supervised distributed message routing scheme implemented in Aparnet [16]. However, the Aparnet scheme was meant to minimize the switching path delays between source and destination. Our scheme is designed to balance load among multicomputer nodes; which has a different set of optimization objectives. Efe and Groselj [7] have also proposed a supervised load sharing model. They use fixed threshold and a controller node, which will execute the extra load transferred from the remaining nodes or transfer them back to the idle nodes for balanced execution. Our scheme differs from their approach in using adaptive thresholds at all nodes and in restricting the host processor to perform only the collection and broadcasting of load indices from all computer nodes. Our host does not participate in the decision of process migration or actual execution of user jobs, which are entirely distributed to local nodes. Our scheme does not require load information exchange among nodes as done in [15]. The distributed decision in our method is based on sender initiation, but using an adaptive threshold which requires no handshaking among nodes as suggested by Krueger and Finkel [9]. Our approach improves from the above methods and greatly reduces the implementation and control overheads, which leads to high performance at lower cost.

The performance of the proposed new method was evaluated by the parellel event_driven simulation [21]. In this paper, we present a dynamic load balancer, which implements the load balancing mehtod, on a 32-node iPSC/386 hypercube multicomputer at USC. The load balancer is written in *C* language and built on top of the NX/2 OS in each i386 node. Benchmark programs are executed in parallel at the process control level. By inserting *run* and *suspend* OS directives to source programs, the invocation of each C function call will

create a new process. Each process is allocated to create child processes or be suspended. The dynamic load balancer averages the system load by process migration. We have performed extensive benchmark experiments to verify the effectiveness of this load balancing scheme.

The rest of the paper is organized as follows: Section 2 describes the adaptive model for distributed load balancing under supervision. Four heuristic migration methods are presented in Section 3. We then describe how to translate sequential programs into the parallel versions in Section 4. The dynamic load balancer developed on iPSC/386 is described in Section 5. In Section 6, we evaluate the performance of those heuristic methods by showing the speedups obtained from executing benchmark programs. The results are analyzed with the corresponding load distributed to all nodes. The final section summaries the research contributions and comments on the impacts of applying the proposed method in executing AI programs, which especially demand load balancing due to unpredictable program behaviors.

2 A New Adaptive Method

We focus our study on multicomputers using point-to-point interconnection networks. A multicomputer is represented by n processor nodes N_i, $0 \leq i < n$, interconnected by a network characterized by a *distance matrix* $D = \{d_{ij}\}$, where d_{ij} shows the number of hops between node N_i and N_j. It is assumed that $d_{ii} = 0$ and $d_{ij} = d_{ji}$ for all i and j. The *immediate neighborhood* of node N_i is defined by the subset $G_i = \{N_j \mid d_{ij} = 1\}$.

A multicomputer is modeled in Fig.2. The *host processor* is connected to all computer nodes. The workload or *load index* l_i is indicated by the number of ready-to-run processes in each node N_i. The load index l_i is passed from each node N_i to the host. The system *load distribution* $L_t = \{l_i \mid 0 \leq i < n\}$ at each time window $(t,\ t + W_t)$, is broadcast by the host to all nodes on a periodic basis. All nodes maintain their own load balancing

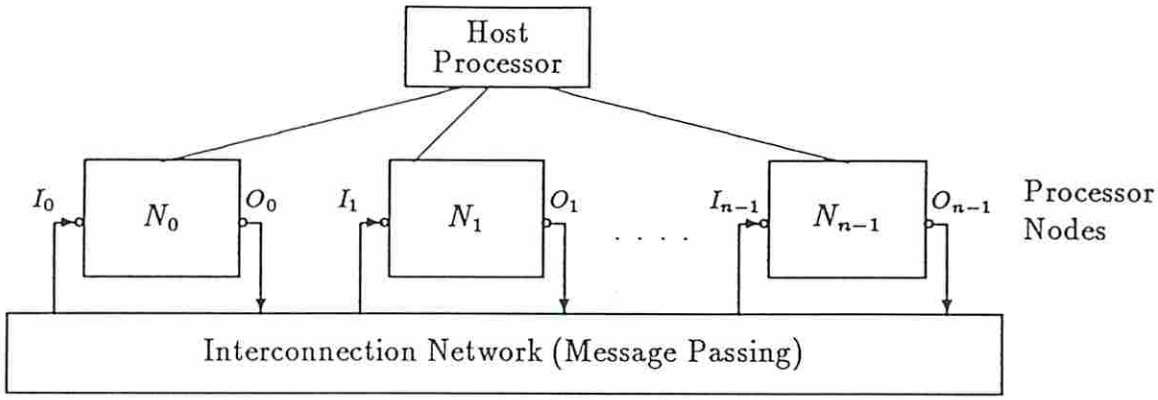


Figure 2: An abstract model of a multicomputer with n processor nodes and a host processor.

operations independently and report their load indices to the host on such a regular basis. The host broadcasts the system status periodically using a variable time windows between tow adjacent updates, depending on traffic situation. The invocation of load balancing activities are totally distributed to all the nodes. Each node N_i has an *input port* I_i and an *output port* O_i. These ports are connected to an interconnection network. Processes created at each node can be either executed locally or migrated via the network to some remote nodes for execution.

The load distribution L_t is described by a *mean value* $\overline{l_t} = \frac{\sum_{i=0}^{n-1} l_i}{n}$ and a *variance* $\sigma(L_t) = \frac{\sum_{i=0}^{n-1}(l_i - \bar{l})}{n}$. Let L_{t_1} and L_{t_2} ($t \geq 1$) be the load distributions at two adjacent update times t_1 and t_2 respectively. The *time window* is defined as $W_{t_2} = t_2 - t_1$. Let r be the *load variation factor* defined by:

$$r = \frac{| \sigma(L_{t_2}) - \sigma(L_{t_1}) |}{max(\sigma(L_{t_1}), \sigma(L_{t_2}))} \tag{1}$$

The parameter r indicates the incremental change in two successive system load distributions. Assume the initial time windows $W_{t_0} = W_{t_1}$. Consider two adjacent update times t_1 and t_2 and choose $0 < k_1 < k_2 < 1$, we compute the time window W_{t_2} from the earlier window W_{t_1}

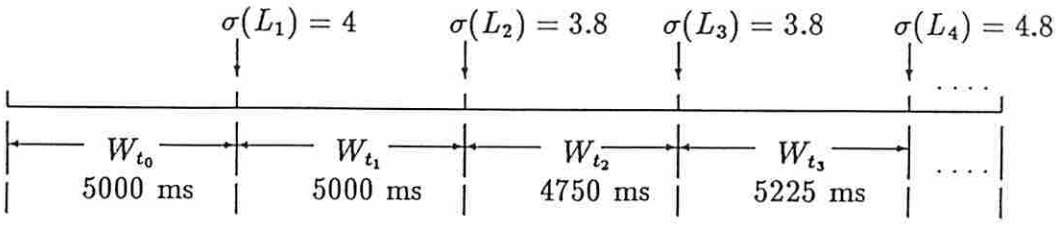


Figure 3: Example variation of the time window function W_t with $k_1 = 0.01$ and $k_2 = 0.1$, note that $W_{t_0} = W_{t_1}$ and W_{t_i} is obtained from $W_{t_{i-1}}$ using Eq.2 for $i \geq 2$.

recursively as follows:

$$W_{t_2} = \begin{cases} (1 - r) \cdot W_{t_1} & \text{if } k_1 \leq r \leq k_2 \\ (1 - k_2) \cdot W_{t_1} & \text{if } r > k_2 \\ (1 + k_1) \cdot W_{t_1} & \text{if } r < k_1 \\ W_{t_1} & \text{if } W_{t_1} < k_2 \cdot W_{t_0} \end{cases} \tag{2}$$

When the system enters a ready state, the system load distribution becomes virtually unchanged. This implies that the time window becomes much longer in a steady state. When the system load changes rapidly, the difference between the load variances becomes significant. Thus the time window will become shorter accordingly. This implies that the system state will be updated more frequently. The parameters k_1 and k_2 are introduced to avoid rapid changes in W_t, especially during the initiation period. In Fig.3, we show an example with initial condition $k_1 = 0.01$, $k_2 = 0.1$ and $W_{t_0} = 5000$ ms. The successive time windows W_{t_i} are calculated from the earlier window $W_{t_{i-1}}$ using Eq.2 recursively.

At each node N_i, we use a sender-initiated load balancing method, where heavily loaded nodes initiate the process migration. The sender-initiated method has the advantage of faster process migration, as soon as the load index of a processor node exceeds certain threshold. We use an *adaptive threshold*, which is updated periodically according to the variation of system load distribution. As modeled in Fig.4, new processes will be put into one of two queues: the *ready queue* R_i or the *migration queue* M_i. Let p be a process, we define a cost

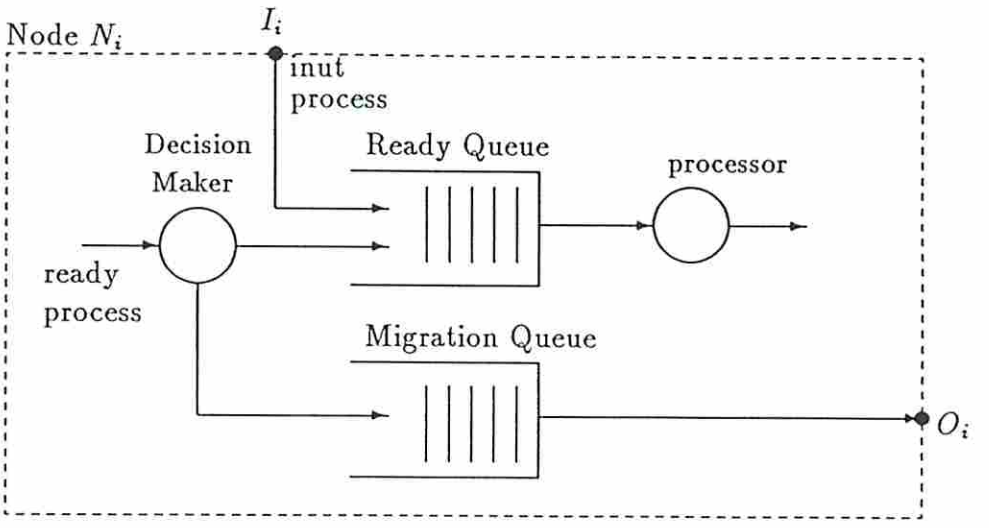


Figure 4: A queueing model describing the operations of distributed load balancer at each processor node.

function $F(p) = e(p) + m(p)$, where $e(p)$ is the *estimated execution time* of p and $m(p)$ is the *estimated memory demand* of p. The values of $e(p)$ and $m(p)$ are estimated at the time of process creation using information available after compile time. The accurate values of $e(p)$ and $m(p)$ can be measured during its life cycle by a monitor. These measures can be used to validate future estimates the same code or with similar behavior. The load index l_i is determined by counting the number of processes in the ready queue R_i at node N_i.

$$l_i = \sum_{p \in R_i} F(p) \tag{3}$$

The variation of the *threshold* δ_i at each node N_i is described in the next section. When a process is ready to run at node N_i, the load index l_i is compared with the local threshold δ_i. If $l_i \leq \delta_i$, the process is put into the ready queue and l_i is incremented by $F(p)$; otherwise it is put into the migration queue. After entering the ready queue, the process will be executed on a the First-Come First-Serve (FCFS) basis. Processes in the migration queue will be migrated to remote nodes eventually. The input port I_i accepts processes migrated from

other nodes and puts them into its ready queue. The output port O_i migrates processes to other nodes.

3 Heuristic Process Migration

Four heuristic methods for migrating processes are formally introduced below. These heuristics are used to invoke the migrating process, to update the threshold, and to choose the destination nodes for process migration. These methods are based on using the load distribution L_t, which varies form time to time. Two attributes are identified below to distinguish the four process migration methods. By considering four combinations of these two attributes, we propose four different heuristic methods for process migration.

1. *Migration and decision range*: The process migration can be restricted to those nodes in the neighborhood set G_i adjacent to each N_i, or involve all the multicomputer nodes in the system. The threshold is updated by either using the *local average load* among neighboring nodes or using the *global average load* among all nodes in the system.

2. *Heuristics used in process migration*: Either a *Round-Robin* (RR) method or a *minimum-load* (ML) method will be used in selecting the destination node for migration. The RR method uses a circular list with a pointer to indicate the front end. The ML method chooses a node with minimum load as the destination node.

A. Localized Round-Robin (LRR) Method

Each node N_i uses the average load among immediate neighboring nodes to update the threshold and migrates processes only to the immediate neighboring nodes from set G_i. The Round-Robin discipline described above is used to select a candidate node for process migration. Two requirements are stated below:

1. After receiving system load distribution L_t from the host at time t, the node N_i resets its threshold δ_i to the average load among immediate neighboring nodes. That is, $\delta_i = \lceil (1+\alpha) \cdot \frac{l_i + \sum_{j \in G_i} l_j}{|G_i|+1} \rceil$, where $0 \leq \alpha \leq 0.2$ is a *normalized constant* chosen.

2. We use an *ordered candidate list* C_i from which we select the destination node. As a set, $C_i = G_i$ and each entry in C_i is a data structure representing a neighboring node of N_i. The entries in C_i are ordered by the increasing load indices involved. Within each time window W_t, C_i is updated in a Round-Robin fashion.

In Table 1, we show an example of using the LRR method in an 8-node hypercube. The normalized constant α is set to be 0.1 in updating each local threshold. The list C_i is increasingly ordered, and the migration queue at each node is initially empty. After a ready process arrives, and the local load index is updated. But the threshold will not be updated until the next L_{t+1} is broadcast from the host. For simplicity, we assume one unit cost for each newly created process. At node N_1, since $l_1 > \delta_1$, it puts the ready process in the migration queue and keeps the load index l_1 unchanged. The process at the front of the migration queue is migrated to node N_3. After that, N_3 is put back to the end of C_1. At node N_3, since $l_3 < \delta_3$, it puts the ready process into the ready queue. Since N_3 also receives 2 processes migrated from nodes N_1 and N_7, its load index l_3 is totally incremented by 3. The candidate list C_3 remains the same because no process needs to be migrated at this point of time.

B. Global Round-Robin (GRR) Method

Each node N_i uses a globally determined threshold and migrates processes to any appropriate node in the system. The selection from candidate list for process migration is again based on the Round-Robin discipline. After receiving the load distribution L_t from

Table 1: Example Load Distributions Before and After the Application of the Local Round-Robin Method

Time	N_i	N_0	N_1	N_2	N_3	N_4	N_5	N_6	N_7
B	l_i	2	10	8	1	6	3	5	15
E	δ_i	8	5	5	10	5	10	10	7
F		N_4	N_3	N_0	N_2	N_0	N_4	N_4	N_3
O	C_i	N_2	N_0	N_3	N_1	N_5	N_1	N_2	N_5
R		N_1	N_5	N_6	N_7	N_6	N_7	N_7	N_6
E									
A	l_i	5	10	8	4	6	4	6	15
F		N_4	N_0	N_3	N_2	N_5	N_4	N_4	N_5
T	C_i	N_2	N_5	N_6	N_1	N_6	N_1	N_2	N_6
E		N_1	N_3	N_0	N_7	N_0	N_7	N_7	N_3
R									

the host, the global threshold δ_t is set to the system average load among all nodes. That is $\delta_t = \lceil (1+\alpha) \cdot \frac{\sum_{j=0}^{n-1} l_j}{n} \rceil$ within the time window W_t. The candidate list C_i operates the same as in the LRR, except $C_i = \{N_j \mid j \neq i\}$.

C. Localized Minimum Load (LML) Method

The way to determine the threshold and to set up migration ports is the same as that in LRR. The difference between LML and LRR methods is in the policy of selecting a destination node. At node N_i, there is a *load table* to store the load index of each node in G_i. The LML method uses the node with the minimum load index in the load table as a destination node. After a process is migrated to the selected node, its load index in the load table is incremented accordingly.

D. Global Minimum Load (GML) Method

In this case, the way to set up the threshold and migration ports is the same as that in the GRR method. But the destination node is determined by finding the node with minimum load on a global basis. That is the node with a minimum load index in the global load table will be selected as the destination node. The GML method requires more computation overhead to search for the node with global minimal load index. With the help of the host processor, this overhead can be minimized and becomes almost negligible.

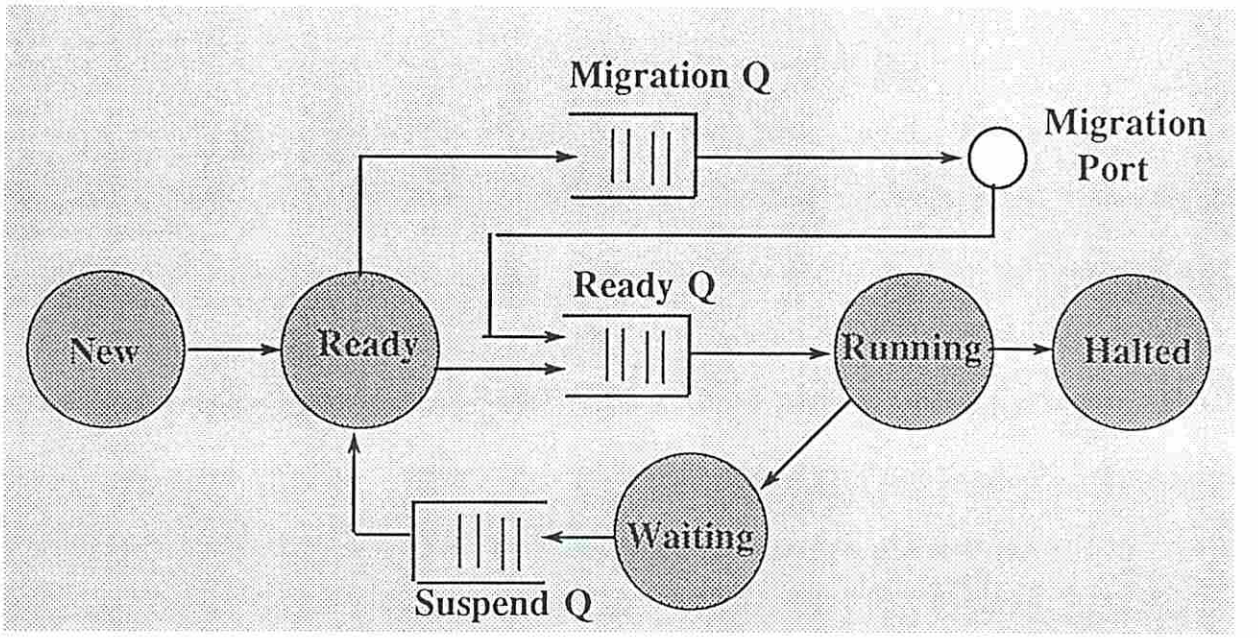
4 Operating System Support

We exploit parallelism in user programs at the process control level, where each process is considered an atomic execution unit. A process is represented by a *Process Control Block* (PCB). Each process can be executed at the creating node, or be migrated to a remote node for execution. The creation and suspension of a process is controlled by two operating system directives: *run* and *suspend*. In this section, we describe the state transition, process control block, and the use of PCBs, *run* and *suspend* OS directives.

Each process may be in one of five states: *new*, *ready*, *running*, *waiting* and *halted*. The state transitions are implemented with three queues: *ready*, *suspend* and *migration* queues. Figure 5 (a) shows state transitions among these queues. The process in the ready queue will be dispatched for execution. When a process is suspended, the process enters the waiting state in the suspend queue. A suspended process can be awakened and becomes ready again. Processes in the migration queue will be transferred to remote nodes, and processes migrated from other nodes will be put into the ready queue.

A. Process Control Block

There are two techniques that can be used for process migration: *complete copying*



(a) Process transitions among various queues

PID (pid,pnode)
Parent PID
Port ID
Executable Code Address
Number of Arguments
Argument Counter
Argument Array

(b) Component fields in a Process Control Block (PCB)

Figure 5: State transitions of a process and the structure of a process control block.

and *code migration.* In complete copying, all copies of user codes have to be allocated to each processor node. The operating system is identically distributed to each node. Each process generated in a user program is represented by a *Process Control Block* (PCB). It is the PCB that will be migrated rather the actual program code. In a code migration approach, different user codes are distributed to disjoint nodes. When a process is migrated, its actual executable code has to be passed to a destination. The complete copying has an advantage of lower migration cost, but requires higher memory demands. It is suitable for a homogeneous multicomputer system. The code migration has lower memory demands but higher process migration cost. There is a tradeoff between these two methods. For parallel processing with a multicomputer, it is often to duplicate application code and distribute them to all nodes working on different data sets. In this work, we will use the complete copying approach.

The structure of a PCB is shown in Fig.5 (b). A process is identified by its Process IDentification (PID), denoted by *(pid,pnode).* The *pid* is an integer and the *pnode* is the parent processor node where a process is created. Thus a PID is a unique identifier of each process. The parent PID identifies the process which creates this process. The port ID determines the port where the results of a process should return to its parent. When a process is halted, the return value along with its parent PID and port ID, is stored in a data structure called *Process_Halting_Result* (PHR). By the semantics of the OS directives defined below, each process is created at a node where its parent process is suspended. Thus a PHR is always returned to its pnode by message passing. At the *pnode*, the parent process in the suspend queue is identified by the parent PID. The actual value is returned to the appropriate port of the parent process. The argument counter counts the number of returns required from the child processes. A process is ready when its argument counter indicates

all the required arguments are available. The argument array stores the data set required by a process. In our process migration scheme, the required data set is migrated within the PCB to the destination node.

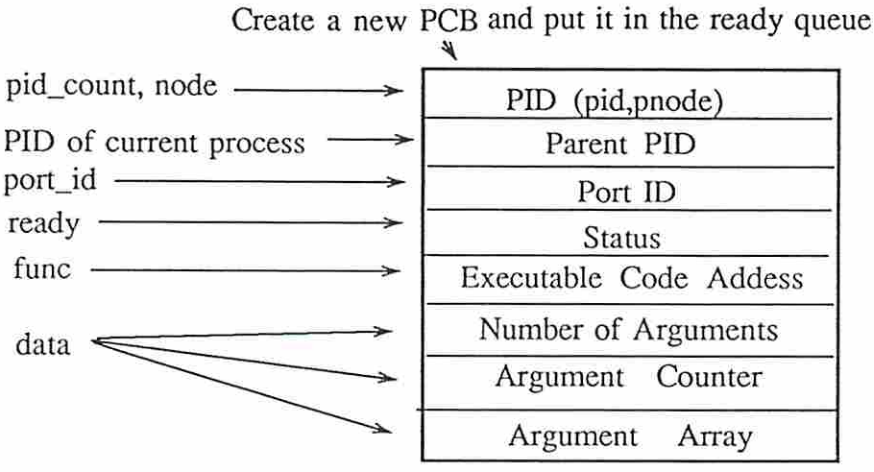
B. Operating System Directives

We modified the *run* and *suspend* OS directives, originally suggested by Chowkwanyun to exploit parallelism in concurrent Lisp program execution [3], to a UNIX/C programming environment. The *run* creates a new process, and the *suspend* suspends an existing process. When a process is dispatched from the ready queue to run, it becomes a *current process.* By inserting *run* and *suspend* into a user program, the running current process can create several new processes for parallel execution. A process is always created at a node where its parent process is suspended. When a process is dispatched from the ready queue to run, it becomes a *current process.* By inserting *run* and *suspend* into a user program, the running current process can create new child processes then be suspended. A process is always created at a node where its parent process is suspended. The semantics of the *run* and *suspend* primitives are show in Fig.6.

- The *run* Primitive

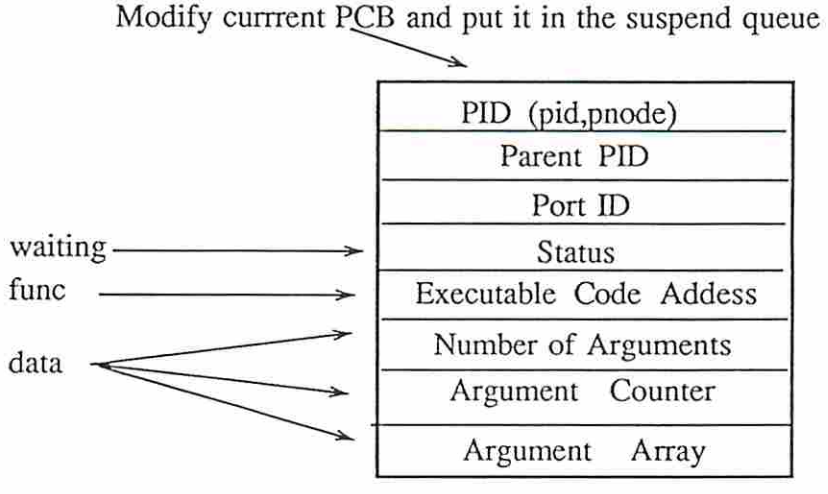
  The *run* primitive creates a new process by assigning it a PCB in the ready queue or the migration queue as shown in Fig.6 (a). The current process is the parent of this newly created child process. The *port_id* identifies the return port at the parent process. The *data* specifies the arguments to run with this process. The actual arguments are passed within the *data.* The PCB is set to be ready, and the number of available arguments matches with the number of arguments required for the process. The *run* primitive does not execute the process immediately but only creates a ready process.

run(func, port_id, data)



(a) Semantics of *run*

suspend(func,data)



(b) Semantics of *suspend*

Figure 6: The semantics of the *run* and *suspend* OS primitives.

execution by calling *run*(fib,0,1,4), where "1,4" indicates the number of available arguments and the actual argument value used, respectively. The process creation tree is used to illustrate the control flow of processes. The actual implementation is represented by a set of PCBs as shown in the example. The invocation of *run*(fib,0,1,4) creates the first process p(0,0) to run with pid = 0 and pnode = 0. The process p(0,0) is suspended, waiting for two additional data arguments. Two child processes p(1,0) and p(2,0) are then created. When these two child processes halt, the results will become the new arguments of "plus()" for the execution of the suspended p(0,0). A similar situation happens to p(1,0). The pair (pid,pnode) is used to identify the process during the migration process.

C. Control Level Parallelism

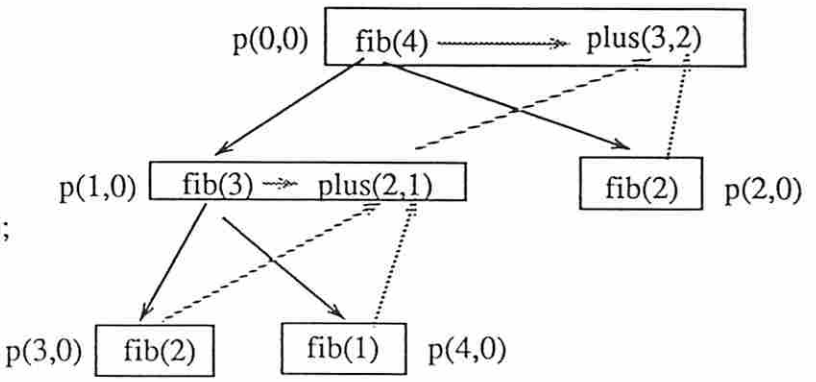
By inserting the *run* and *suspend* primitives into a user program, each major function call is treated as a process. When recursive call occurs, the execution of the current process is suspended and a new child process is created. This was illustrated by the example in Fig.7. Without process migration, each process is created, executed and suspended at the local processor node. In a multicomputer system, parallelism can only be achieved by static allocation which distributes computation to each processor node. If the run time conditions are unpredictable, then the operating system has no control to balance the workload.

For the example shown in Fig.1, the contents of the ready and suspend queues are far from balanced without applying load balancing (Fig.8 (a)). There are 12 processes waiting for execution at node N_1 and only 1 processes in the ready queue at node N_0. Clearly, the total execution is bounded by the time at node N_i. Therefore, parallelism cannot be fully exploited in this case. By applying the process migration technique, processes can be migrated to remote nodes for execution. The variations of the contents in the ready and

Sequential Program

```
int
fib(x)
int x;
{
  if ( x <= 2 )
     return(x);
  else
    return(fib(x-1) + fib(x-2));
}
```

Process Creation Tree

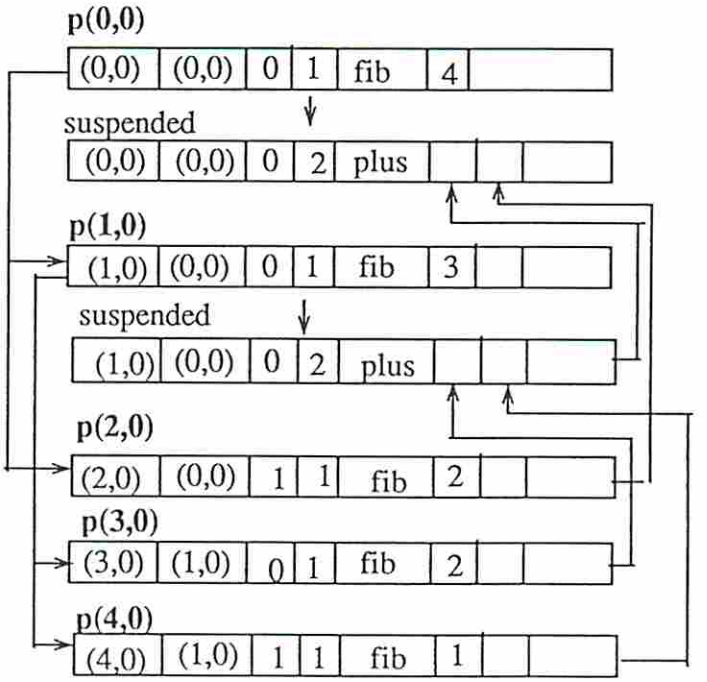


Parallel Program

```
int
fib(x)
int  x;
{
   int     suspend();
   int     run();
   int      plus();
   int      fib();

   if ( x <= 2 )
     return(x);
   else {
    suspend (plus,0,2);
    run(fib,0,1,x-1);
    run (fib,1,1,x-2);
   return(NULL);
  }
```

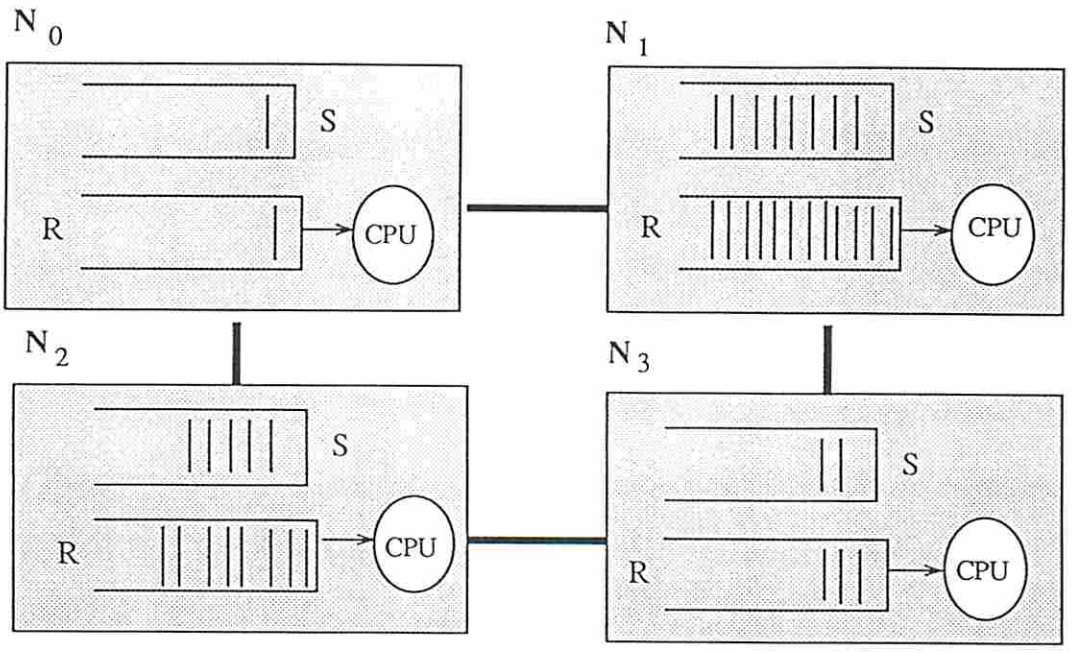
Process Control Blocks



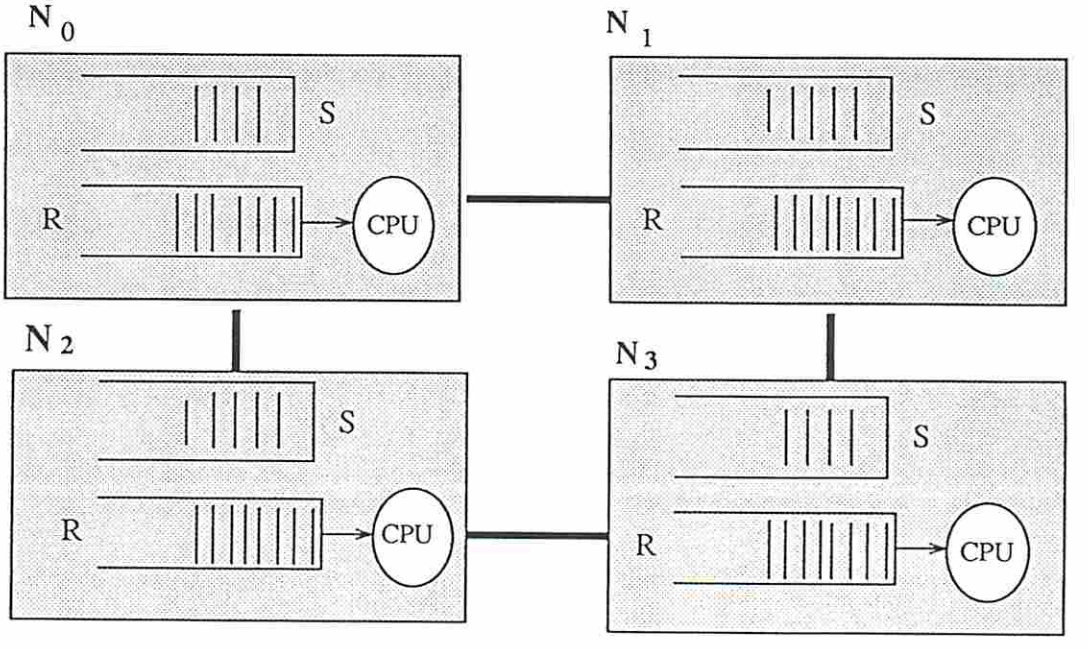
Process Invocation

run(fib,0,1,4);

Figure 7: The process creation tree and the PCBs used in the invocation of the *fibonacci* function *fib*(4), at node 0 of a 32-node multicomputer system.



(a) Unbalanced without process migration.



(b) Balanced with process migration.

Figure 8: Exploiting parallelism by process migration among 4 computer nodes as examplified in Fig.1 (R: ready queue, S: suspend queue).

suspend queues are shown Fig.8 (b). These queues at four nodes will end up a balanced distribution of having almost equal number of processes in each. The purpose is to keep all nodes busy during the entire execution time. Note that the processes at the suspend queue can be awakened, when results are returned from child processes.

5 Implementation of the Load Balancer

To show the effectiveness of the proposed heuristic load balancing methods, we have implemented a prototyping dynamic load balancer on a 32-node hypercube system. The load balancer is written in C using iPSC/386 message passing library calls. It consists of a host program and many distributed nodal programs. The host program is executed at the host processor with Unix. The nodal programs are executed i386 processor nodes on using the NX/2 operating system. We first describe the construction of the load balancer, then discuss the message passing mechanisms used in the implementation.

The host program consists of a *loader*, a *window adjuster*, a *load information updater*, an *asynchronous mailman* and a *reporter* as shown in Fig.9. The *loader* loads in the load balancers and distributes partitioned user programs into local nodes. The *time window adjuster* updates the time window with inputs from the load information updater. The *load information updater* periodically collects system load distribution from all nodes, severing the host processor. The *reporter* displays the necessary messages during the program execution and reports the user program execution result and the performance data. When all expected results of a user program are received from distributed nodes, the *reporter* sends a stop message to each node. Then the *kernel* at each node will stop the execution. All message passing to/from distributed nodes are coordinated by an asynchronous mailman in the load balancer.

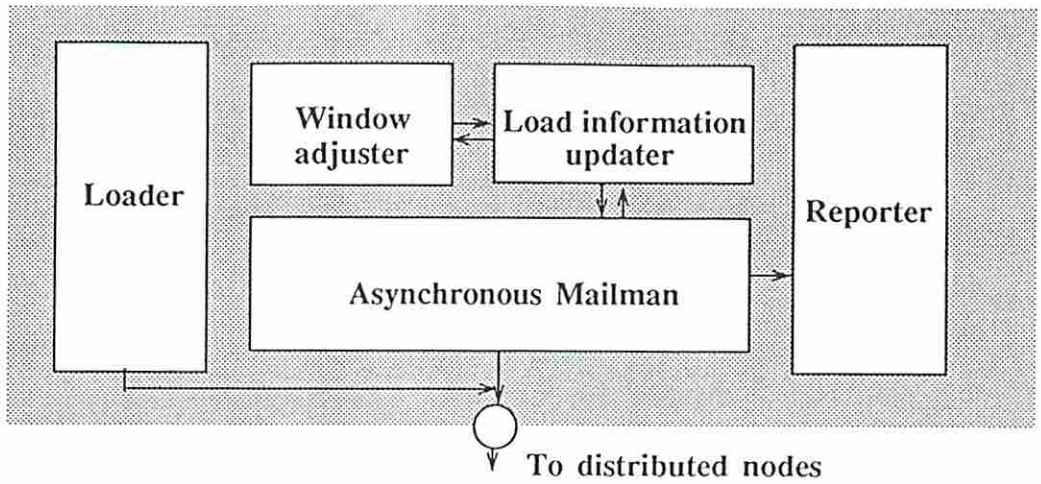


Figure 9: The supervision kernel in the host processor.

The distributed load balancers are identical among all nodes. Figure 10 shows the construction of the load balancer and the process flow under its control. The *kernel* is the control module of the load balancer. It initiates the execution of a user program by creating an initial process and assigning processor to it. The kernel then repeatedly calls the *dispatcher*, the *load transfer* and the *mailman*, until receiving a stop message from the host. The *mailman* sends and receives messages asynchronously. A message can be a request for local load index l_i from the host, the local load index l_i reporting to the host, the system load distribution L_i broadcast from the host, a migrated processes from/to the remote nodes, a return result of a halted process from/to remote nodes, and display or debug messages to the host.

The execution of a process may end up with one of the following cases: 1) being halted with a return result, 2) being suspended, 3) creating some new child processes and being suspended. A newly created or an awaken process is in the ready state. The *decision maker* uses updated threshold δ_i to decide whether a ready process should be put into the ready queue or migration queue. A process will be dispatched by the *dispatcher* and get the processor for its execution. The processor scheduling discipline is implemented by the

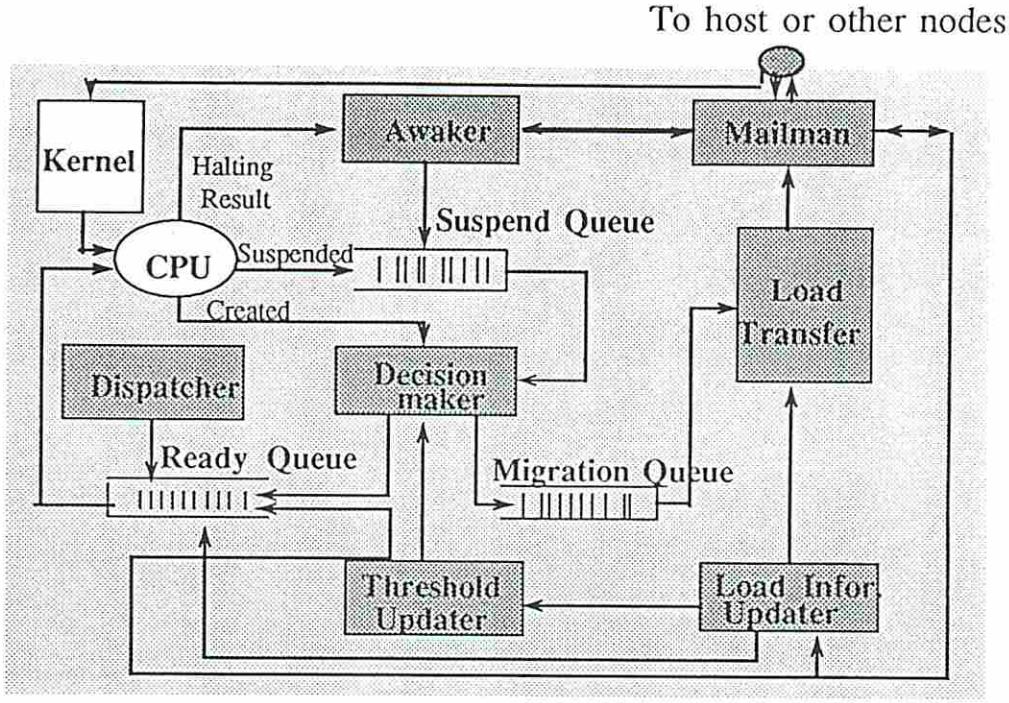


Figure 10: The load balancer program at each node processor.

dispatcher. The *threshold updater* updates the threshold δ_i. The load information updater calculates the local load index l_i by checking the ready queue, sends it to the host through the *mailman* and receives the system load distribution L from the host.

The migration destination is decided by a *load transfer*. It implements the load transfer policy and handles the candidate list and the load table. The halting result of a process execution is sent to an *awaker*. The result can be directly obtained from local process execution or through mailman from the process execution at the remote node. The awaker finds the parent process identified by the parent PID of the result, and puts the return value to the appropriate port. When the argument counter of a suspended process researches the required number, it is changed to a ready state.

Asynchronous message passing mechanism is used in the implementation of the load balancer. We have developed a generic message pattern which can be passed among nodes

easily. A generic message has its own type defined as G_TYPE. It is a data structure consisting of a *type*, a *header* and a *body*. The type is a flag to identify the kind of actual message. The header specifies the source, destination and the length of the message body. The body can be a PCB, a computational result, a system load distribution L, a node load index l_i, etc; each is represented by a dedicated data structure. The service routine *irecv*() is invoked within the control loops at host and node load balancers. It checks to see if there is any generic message arrivals. Upon the arrival of a message, the routine *crecv*() is called to receive the message. By checking the message type, appropriate actions will be taken. Using this asynchronous mechanism, the distributed load balancing at each node acts independently. Each node works on its own without waiting for responses from other nodes.

6 Experimental Benchmark Results

The performance of four heuristic methods for load balancing was evaluated by parallel execution of selected benchmark programs on a 32-node iPSC/386 hypercube system. In this section, we describe the computational and communication characteristics of those benchmark programs used, report the software experiments performed, and then conclude on the performance results obtained.

A. Benchmark Programs

The benchmark programs chosen have the following common characteristics: 1) *strong data dependency on the argument set used* and 2) *unpredictable program behavior at run-time*. These are C programs inserted with *run* and *suspend* OS directives for explicit control of parallel activities. We have implemented three AI-related programs namely: the *tak*, the *fibonacci* and the *n_queen* programs. The programs *tak* and *fibonacci* are recursive

functions, where *tak* takes three arguments and *fibonacci* uses only one argument. The *N_queen* is a nondeterministic backtracking search program, using a one-dimensional data array. The execution times of these programs are all data dependent, and their behaviors are not predictable at compile time. This phenomenon can be seen from examining the following sample programs with different argument values.

- *tak*
  tak(18,16,9) needs 11842 function calls.
  tak(18,16,15) needs 7 function calls.
- *fibonacci*
  fibonacci(20) needs 13529 function function calls.
  fibonacci(3) needs 3 function calls.
- *n_queen*
  n_queen(10) needs 34815 backtracking searches and has 724 solutions.
  n_queen(4) needs 15 backtracking searches and has 2 solutions.

The benchmark program *fibonacci fib*() was parallelized as illustrated earlier in Fig.7. Programs *tak* and *n_queen* can be written in a similar fashion. Using the OS support introduced in Section 4, these programs generate different process invocation trees depending on the arguments used. Both the *tak* and *fib* programs calculate the summation of a set of functions. The *n_queen* program counts all possible results from a set of nondeterministic searches. The execution of each function call or each search call is initially assigned to one node in the multicomputer. After each node finishes its computation, the result is reported to the host. The host has to receive results from all nodes, before summing them up at the very end. We purposely chose special data arguments causing very unbalanced process creations at various nodes. Two types of experiments were carried out. Let n be the number of processor nodes used in a multicomputer. Our experiments cover machine sizes ranging from 2 to 32 processor nodes.

Case 1: Extremely Unbalanced

- *Tak*: calculate $X = \sum_{i=0}^{n-2}$tak(18,16,15)+tak(18,16,9)
  - Execute *tak*(18,16,9) at node N_1, (11842 calls)
  - Execute *tak*(18,16,15) node N_i, $i \neq 1$, (7 calls)
- *Fibonacci*: calculate $X = \sum_{i=0}^{n-2}$fib(3)+fib(20)
  - Execute $fib(20)$ at node N_1, (13529 calls)
  - Execute $fib(3)$ at node N_i, $i \neq 1$, (3 calls)
- *N_queen*: calculate the $X = \sum_{i=0}^{n-2}$n_queen(4)+n_queen(10),

  where *n_queen*() returns the number of possible solutions.
  - Execute *n_queen*(10) at node N_1, (34815 searches)
  - Execute *n_queen*(4) at node N_i, $i \neq 1$, (15 searches)

Case 2: Random Load Distribution

- *Tak*: calculate $X = \sum_{i=0}^{n-1} tak(18,16,j)$, $9 \leq j \leq 15$

  Execute *tak*(18,16,j) at each node N_i, such that $j = random(9,15)$
- *Fibonacci* calculate $X = \sum_{i=0}^{n-1} fib(j)$, $1 \leq j \leq 20$

  Execute fib(j) at each node N_i, such that $j = random(1,20)$
- *N_queen* calculate $X = \sum_{i=0}^{n-1} n\_queen(j)$, $4 \leq j \leq 10$

  Execute *n_queen*(j) at each node N_i, such that $j = random(4,10)$

In Case 1, special arguments were chosen to yield extremely unbalanced load distributions. Case 2 uses some randomly generated load distributions. Without load balancing,

the total execution time required is dominated by the node creating the maximum number of processes. Let T_i be the compute time of each node N_i. Let S_n be the *speedup factor* of using n processor nodes defined as:

$$S_n = \frac{P_1}{P_n} = \frac{\sum_{i=0}^{n-1} T_i}{max(T_i)} \tag{4}$$

where P_1 is the time to execute the program on one processor node and P_n is the time to execute the same program on a n-node multicomputer. Since the atomic processes created in these benchmark program execution are homogeneous with the same executable code and the same memory demand, the execution time $e(p)$ and the memory demand $m(p)$ of each process p will be the same. Thus T_i can be approximated as $T_i = e(p)\times$ (the number of processes executed at N_i). We first analyze the these two Cases without any load balancing.

For the experiments in Case 1, node N_1 will create a large number of processes, while other nodes will create only a few. After all nodes other than N_1 pass their computational results to the host, the host must wait for N_1 to finish its execution before summing up all the results. For the example $fib()$ program, obviously $T_1 \gg T_{i\neq 1}$, since $T_1 = 13529 \cdot e(p)$ and $T_i = 3 \cdot e(p)$. Thus we can write

$$S_n = \frac{\sum_{i\neq 1} T_i + T_1}{T_1} = \frac{(n-1)\cdot 3 \cdot p(e) + 13529 \cdot p(e)}{13529 \cdot p(e)} = \frac{(n-1)\cdot 3 + 13529}{13529} \doteq 1 \tag{5}$$

This indicates the fact that no appreciable speedup can be achieved in Case 1 experiments if load balancing is not applied.

For those experiments in Case 2, the speedup S_n depends on the randomness of data arguments generated. Since the number of processes created at each node N_i is dependent on the argument value, the computing time T_i also depends on the randomly generated data argument. Therefore $S_n = \frac{\sum T_i}{max(T_i)}$ will become linearly proportional to n.

We assume a unit time for executing each atomic process. The load index at node N_i is thus calculated as $l_i = |R_i|$, the cardinality of the set R_i in the ready queue. The initial time window W_0 is set to be 2 sec before the first update, and the constants used are: $k_1 = 0.001$, $k_2 = 0.1$ and $\alpha = 0.1$. Now, we are ready to analyze the performance timing results obtained in benchmark runs.

B. Experimental Results

The speedup factor is calculated as $s_n = \frac{P_1}{P_n}$ using the actual values of P_1 and P_n measured from parallel execution of benchmark programs. The speedup factor measures the actual performance improvement achieved. The effectiveness of load balancing is analyzed based on the actual number of processes executed at various nodes. The four heuristic methods are compared with the NLB (No Load Balancing) and the GRD (GRaDient) methods, in order to show their relative merits. By NLB, we mean that processes created at each node must be executed locally without process migration. The GRD method is based on sender-initiation with a fixed threshold as originally proposed by Lin and Keller [11]. This method requires frequent load index exchange among communicating nodes. The method forms a gradient in which the loads are always transferred from busier nodes to idle ones. However the GRD method is ineffective when the system is heavily loaded, since the fixed threshold will continue swapping the load almost without stop.

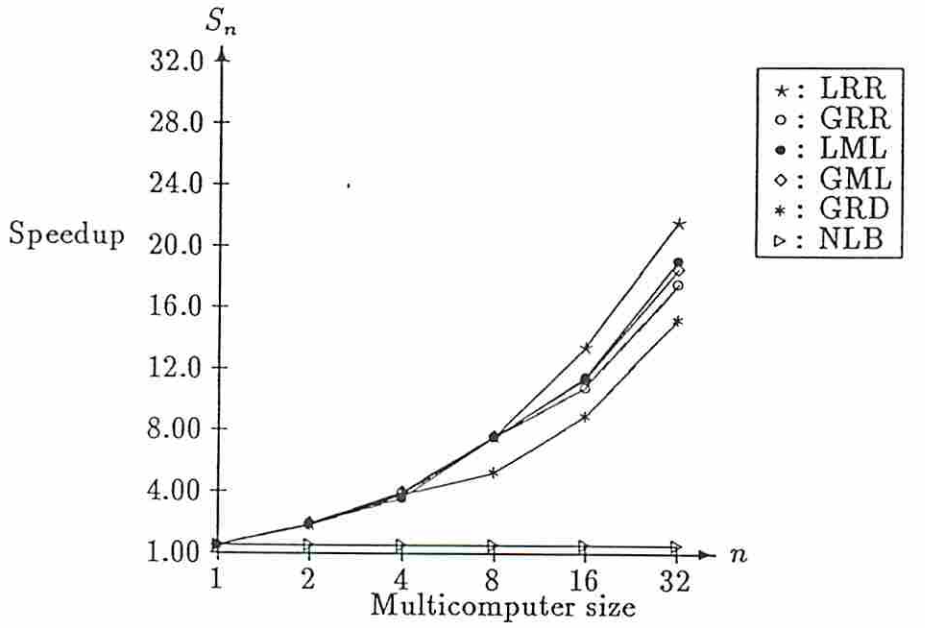
The results from Case 1 experiments are shown in Parts (a) of Figs. 11-13. Without load balancing, there is no speedup ($S_n = 1$). As analyzed before, this is due to the fact that the total execution time is determined by the execution time at node N_1. The speedup obtained from using four heuristic methods increases sublinearly as the multicomputer increase its size. Comparing with the GRD method, our heuristic methods are superior. The GRD

method has linear speedup, when the system has fewer than four processor nodes. When the system size becomes larger, it performs worse than any of the heuristic methods used.

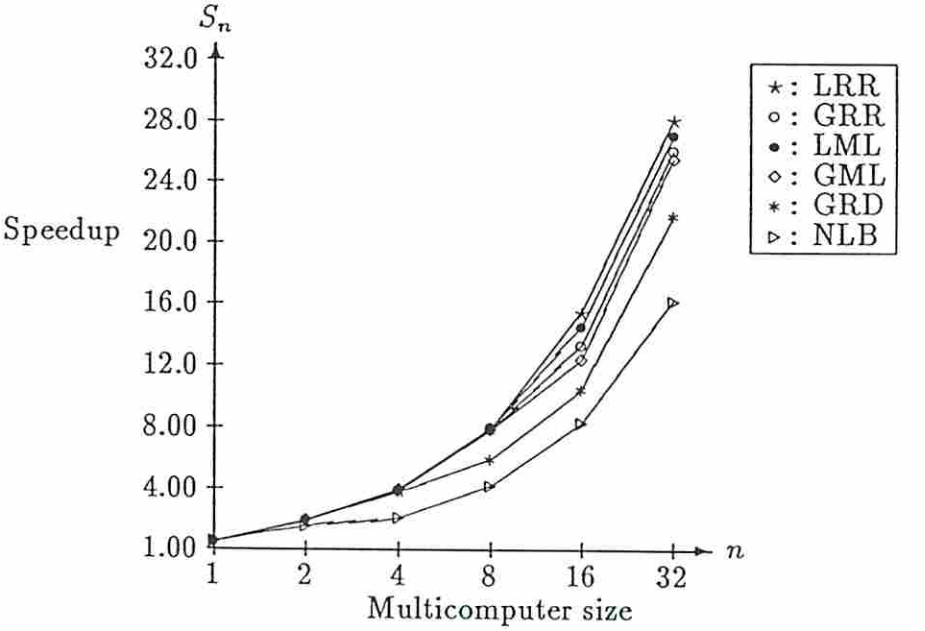
The results from Case 2 experiments are shown in Parts (b) of Figs. 11-13. Being different from Case 1, moderate speedups are obtained even without load balancing. This is due to the random distribution of the data sets. The total compute time is still determined by the node which executes most of the processes. For example, if tak(18,16,9) is executed at node N_4 and tak(18,16,10) is executed at node N_0, then node N_4 executes most of the processes, and the compute time at node N_0 becomes immaterial. The speedup is improved significantly, when dynamic load balancing is applied. The improvement was obviously obtained by reducing the compute time at those heavily loaded nodes via heuristic process migration.

The *efficiency*, defined by the ratio $\frac{S_n}{n}$, of our load balancing methods varies from 60% to 100% as shown in Fig.14 from executing the $fib()$ program. Similar results can be derived from the speedup curves associated with executing the *tak* and *n_queen* programs. The efficiency decreases with occasional fluctuations, as the multicomputer size increases. However, our methods maintain a 60% lower bound. The NLB method shows a very efficiency as expected.

A snap short of the load distribution in 32 nodes is shown in Table 2 for the balanced execution of the three benchmark programs. The entries of this table are based on experiments in Case 1, using the LRR method. The load index indicates by the number of processes actually executed at each node. Let $\overline{x}$, σ and μ be the *mean value*, the *variance* and the *standard deviation* of the load distribution to all nodes respectively. Since we use the number of processes resident in the ready queue as the load index, the system load distribution is relatively balanced as demonstrated by the table entries. Note that if only one processor node is used, all spawned processes have to be executed at that node. Therefore the summation

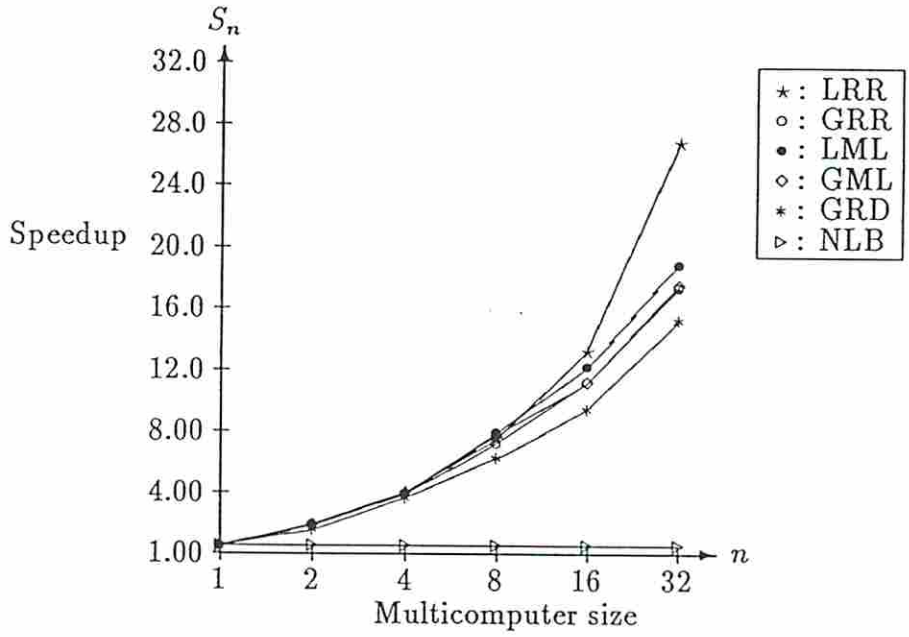


(a) Case 1 Experiments

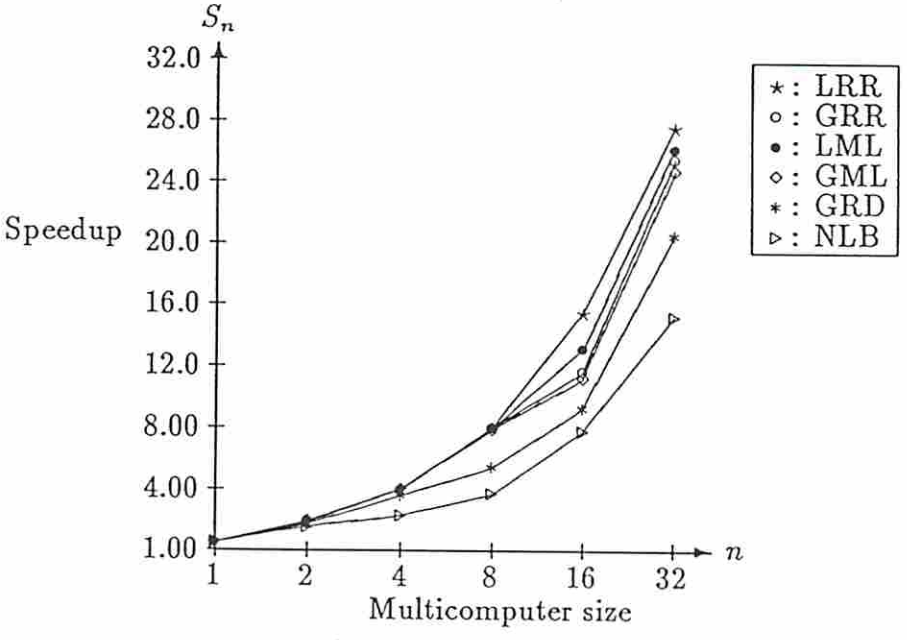


(b) Case 2 Experiments

Figure 11: Speedup obtained from balanced execution of the *tak* program.

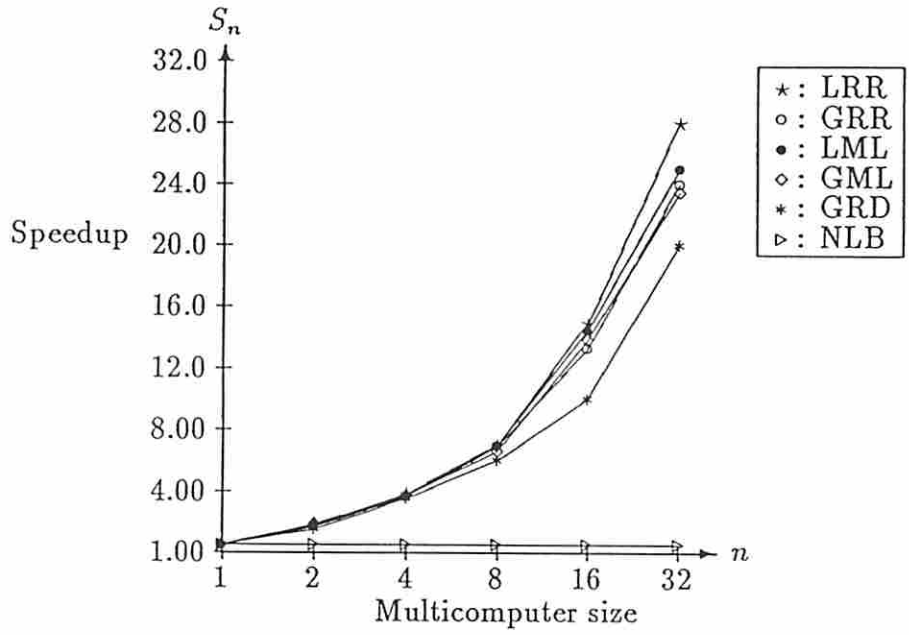


(a) Case 1 Experiments

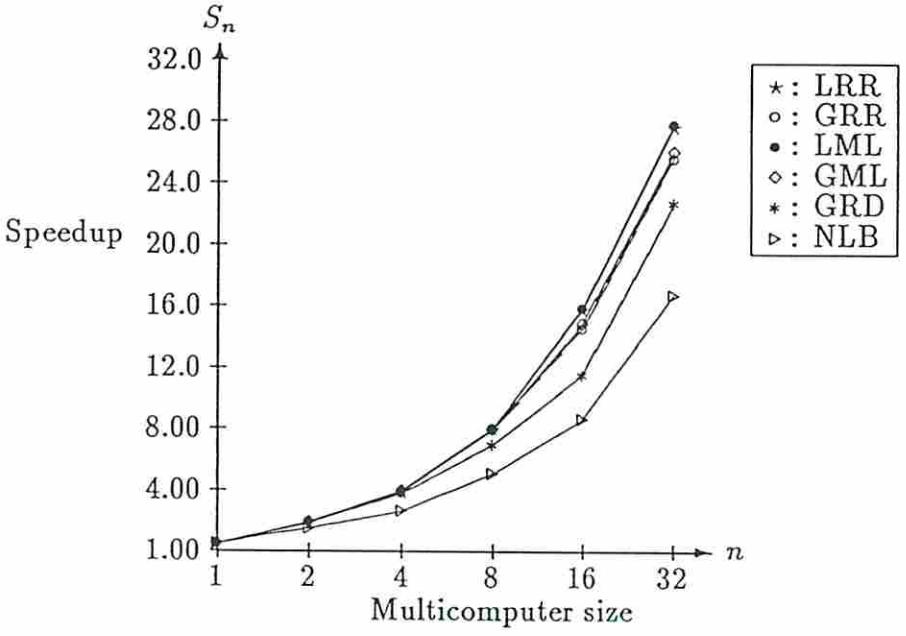


(b) Case 2 Experiments

Figure 12: Speedup obtained from balanced execution of the *fib* program.

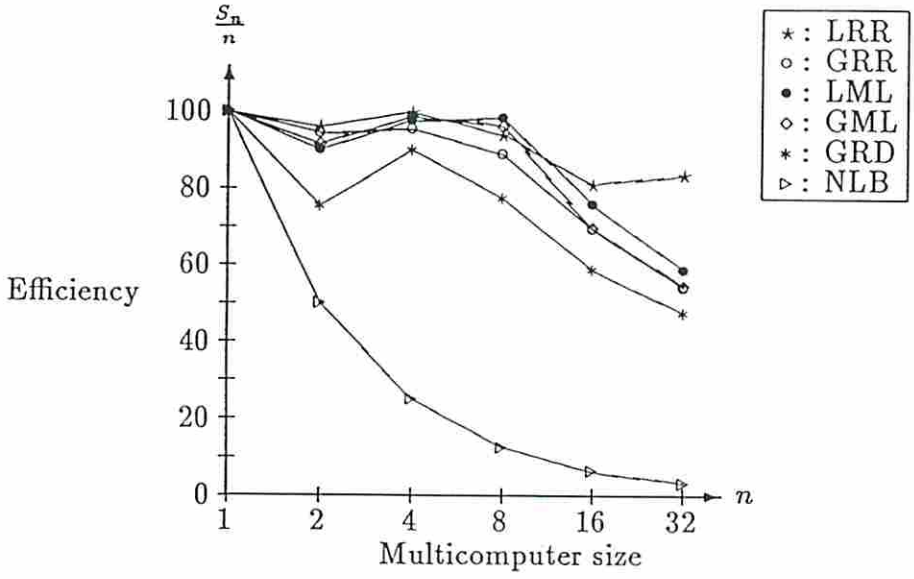


(a) Case 1 Experiments

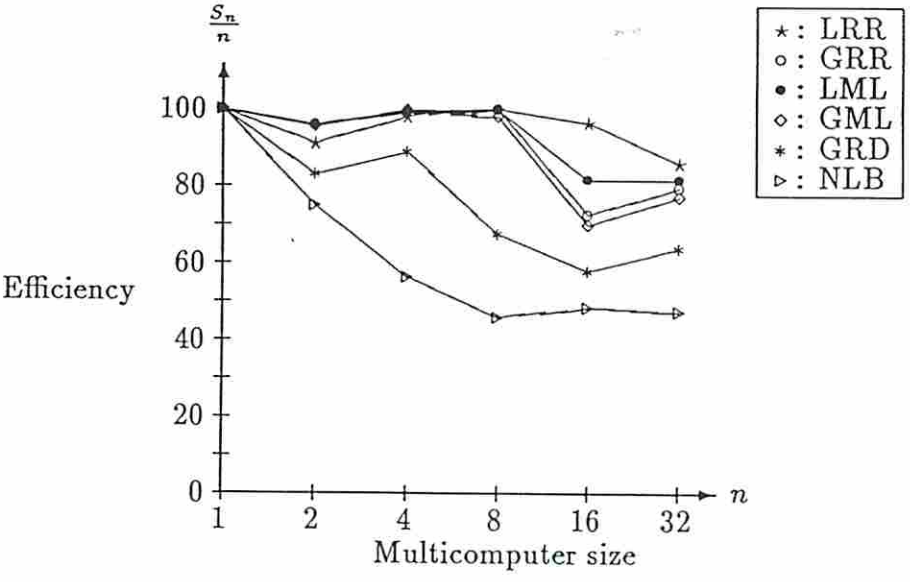


(b) Case 2 Experiments

Figure 13: Speedup obtained from balanced execution of the *n_queen* program.



(a) Case 1 Experiments



(b) Case 2 Experiments

Figure 14: Efficency obtained from balanced execution of the *fib* program.

Table 2: Load Distribution in 32 Nodes for Balanced Execution of Three Benchmark Programs.

Program	*Tak*	*Fibonacci*	*N_queen*
$\overline{x}$	376.8	426	1103
σ	114.1	48	65
μ	10.7	6.9	8.1
N_0	463	483	1115
N_1	688	495	1231
N_2	382	485	1147
N_3	583	473	1199
N_4	388	511	1251
N_5	523	433	1133
N_6	361	473	1164
N_7	367	389	1037
N_8	376	505	1204
N_9	496	451	1149
N_{10}	454	503	1155
N_{11}	382	405	1096
N_{12}	379	407	1103
N_{13}	448	411	1000
N_{14}	427	363	1160
N_{15}	358	385	1025
N_{16}	346	505	1101
N_{17}	601	401	1044
N_{18}	340	383	1141
N_{19}	319	407	1105
N_{20}	304	359	1024
N_{21}	349	375	1009
N_{22}	322	357	1090
N_{23}	307	461	1094
N_{24}	262	407	1099
N_{25}	304	431	1092
N_{26}	253	391	1080
N_{27}	388	375	1038
N_{28}	262	421	1039
N_{29}	307	403	1132
N_{30}	115	383	1017
N_{31}	205	391	1039

of all processes executed at all nodes divided by the number of processes executed at the busiest node gives a first-order approximation of the speedup expected. However, the actual speedups measured by the execution times are higher than those guessed values. This is due to the FCFS processor scheduling used, where the queue manipulation takes longer time to complete for longer queue length. We view this only negligible constant an implementation overhead. The results we obtained verify those heuristic dynamic load balancing methods do perform better with low implementation overhead.

There exist no drastic differences in performance among the four heuristic methods. Two explanations are given: 1) The system size n=32 used in the experiments is too small to make a difference. 2) The hypercube network is restricted to only point-to-point communications. As far as locality is concerned, the local migration methods (LRR and LML) are better than the global migration methods (GRR and GML) for point-to-point network as the system size grows. The global methods demand a higher cost in process migration in large point-to-point systems. For a multiple-bus network, the communication cost between two nodes is independent on the physical distance between them. Therefore, the global methods may perform better, since the load is more balanced on a global basis.

As far as the use of heuristics is concerned, the Round-Robin methods (LRR and GRR) perform better, when the system size becomes larger. Both minimal-load methods (LML and GML) are better for smaller system sizes. Basically, the minimum load methods use more accurate information in selecting the destination node than the Round-Robin methods do. However, the overhead to find a proper destination node with the minimal load becomes higher, when the system size becomes very large. There is a tradeoff between the accuracy desired and the implementation cost. In summary, the choice among these methods is sensitive to the size of the multicomputer, the interconnection topology, and the implemen-

tation costs incurred. All of these factors contribute to the efficiency of the programming environment to be used.

7 Conclusions

We have proposed four heuristic methods for dynamic load balancing in a multicomputer system. These methods require less control overhead as compared with previously proposed methods. This is accomplished by using a host processor to collect system load information and to update the load balancing threshold on a periodic basis. Since the host operations are overlapped with distributed executions at multiple nodes, the system overhead is merged into the actual compute time. The threshold used by those heuristics is adaptively adjusted using the most recently updated system information. We use either the Round-Robin discipline or a minimum-load strategy to select the destination node for process migration. The performance of these methods is evaluated by parallel execution of representative benchmark programs. These benchmark programs reflect some behavior of AI-related operations.

A dynamic load balancer has been implemented on a iPSC/386 hypercube system. Benchmark programs are parallelized at the process control level by using the *run* and *suspend* OS directives. The speedup after parallel execution is achieved by applying dynamic load balancing. Although these experiments are performed on a hypercube architecture, the proposed methods are implementable on other network topologies as well. The relative merit of each method can be exploited on various multicomputer architectures. One can make an intelligent and cost-effective choice among the four methods based on special communication characteristics in a multicomputer. Because most AI-oriented programs have an unpredictable run-time behavior, distributed AI processing is hard to achieve without dynamic load balancing. Our benchmark experiments verify the effectiveness of using any of

the four heuristic methods for dynamic process migration, especially applying to the domain of distributed AI applications. The particular choice among the four methods would be more a concern of cost-effectiveness than an academic one. In an earlier paper [20], we presented a static load balancing method for mapping rule-based production systems. This present paper complements the earlier effort by solving the dynamic load balancing problem for distributed AI processing at run time.

References

[1] K.M. Baumgatner and B.W. Wah, "GAMMON: A Load Balancing Strategy on Local Computer System with Multiaccess Networks", *IEEE Trans. on Computers*, pp.1098-1109, vol. 38, no. 8, August, 1989

[2] T.L. Casavant and J.G. Kuhl, "A Formal Model of Distributed Decision Making and Its Application to Distributed Load Balancing", *Proceedings of International Distributed Computing*, pp. 232-239, August, 1986.

[3] R. Chowkwanyun, "Dynamic Load Balancing In Concurrent LISP Execution on a Multicomputer System", *Ph.D. Disseratation*, Dept. of EE Systems, Univ. of Southern California. 1988.

[4] George Cybenko, "Dynamic Load Balancing for Distributed Memory Multiprocessors", *Journal of Parallel and Distributed Computing* pp. 279-301, vol.7 no. 2, October, 1989

[5] T.C. Chou and J.A. Abraham, "Distributed Control of Computer Systems", *IEEE Trans. on Computers*, vol.C-35, no. 6, June, 1986.

[6] D. Eager, E. Lazowsk, and J.Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Trans. on Software Engineering*, SE-12, No.5, 662-675, 1986

[7] K.Efe and B. Groselj, "Minimizing Control Overhead in Adaptive Load Sharing", *IEEE, Internal Conference on Distributed Computing*, pp. 307-314, 1989

[8] R. Ferrari and S. Zhou, "A load Index for Dynamic Load Balancing", *Proc. ACM-IEEE Fall Joint Comp. Conf.*, pp. 684-690, 1986

[9] P. Krueger and R. Finkel, "An Adaptive Load Balancing Algorithm for a Multicomputer", *Tec. rep 39, University of Wisconsin, Madison*, April 1984.

[10] K. Hwang and D. DeGroot (Eds), *Parallel Processing for Supercomputers and Artificial Intelligence*, McGrow Gill, N.Y. March 1989.

[11] F.C.Lin and R.M.Keller, "Gradient model: a Demand-driven Load Balancing Scheme", *IEEE Proceedings of 6th Conference of Distributed Computing*, pp. 329-336, August, 1986

[12] M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems", *Proceedings ACM Computer Network Performance Symposium*, April, 1982

[13] L.M. Ni and K. Hwang, "Optimal Load Balancing in Multiple Processor System with Many Job Classes", *IEEE Trans. on Software Engineering*, pp. 491-496, vol.SE-11, no. 5, May, 1985.

[14] L.M. Ni, C. Xu, and T.B. Gendreau, "A Distributed Grafting Algorithm for Load Balancing", *IEEE transactions on Software Engineering*, vol. SE-11, No.10, pp.1153-1161, October, 1985.

[15] S. Pulidas, D. Towsley and J. Stankovic "Embedding Gradient Estimators in Load Balancing Algorithms", *IEEE Proceedings of International Conference in Distributed Computing*, pp. 482-490, 1988.

[16] A.S. Tanenbaum, *Computer Networks*, Prentice Hall, Inc., Englewood Cliffs, N.J. 1989.

[17] A.M. Tilborg and L. Wittie, "Wave scheduling - decentralized scheduling for task forces in multicomputers", *IEEE Trans. Computers,* Vol. C-33, pp. 835-844, September, 1984.

[18] B.W. Wah and P. Mehra, "Learning Parallel Search in Load Balancing", *Proceedings Workshop Parallel Algorithms Machine Intelligence Pattern Recognition*, AAAI, Minneapolis,MN, August, 1988

[19] Y. Wang and R. Morris, "Load Sharing in Distributed Systems", *IEEE Trans. on Computers*, pp. 204-217, vol. c-34, no.3, 1985

[20] J.Xu and K. Hwang, "A Simulated Annealing Method for Mapping Production Systems onto Mutlicomputers", *Proceedings of the sixth IEEE Conference on Artificial Intelligence Applications*, pp. 130-136, Santa Barbara, CA. March 7, 1990.

[21] J.Xu and K.Hwang, "Heuristic Methods for Dynamic Load Balancing in a Message-Passing Supercomputer", appear in *Proc. IEEE and ACM Supercomputing '90*, New York, NY. November 12-16, 1990.