Dynamic Load Balancing for Parallel Program Execution on a Message-Passing Multicomputer

Jian Xu
IBM Corporation
5600 Cottle Road
San Jose, CA. 95193

Kai Hwang
Department of Electrical Engineering
University of Southern California
Los Angeles, CA. 90089-0781

Abstract

Dynamic load balancing solves the remapping problem in a multicomputer system at run time, where many processes need to be allocated evenly to multiple processor nodes. The mean is to migrate processes from busy to idle nodes in order to achieve higher resource utilization. We have implemented a distributed load balancer at an iPSC/2 hypercube computer system, which uses heuristic methods to balance the system load adaptively. These heuristic methods cooperate with a central supervisor at the host machine, and invoke the load balancing activities under decentralized control. Benchmark experiments have shown that the proposed dynamic load balancing methods can speed up the parallel execution of benchmark programs significantly.

Introduction

A multicomputer is a multiprocessor system with distributed memories. To achieve parallel processing on a multicomputer, a user program first needs to be partitioned into multiple modules and loaded into distributed processor nodes after compilation. At run time, each node starts with an initial process which may create child processes. For many user programs, the run-time condition is data dependent and unpredictable. Even if starting with a fairly balanced initial allocation, the initial processes at some nodes will create many processes, while the others will create only a few, as shown in Fig.1. To improve the resource utilization, processes created at run-time need to be remapped into processor nodes by dynamic load balancing.

A better dynamic load balancing method should reduce the overhead in collecting load indices and in process migration [2], [3], [6], [9]. In this paper, we propose four heuristic methods for dynamic load balancing, which avoid frequent load information exchanges among nodes by a central supervisor. Processes are migrated to remote nodes are subject to distributed activities. Many proposed load balancing methods use fixed thresholds to determine heavily loaded nodes [1], [4], [5]. If the threshold is too low, the thrashing caused by excessive load balancing activities may degrade the performance significantly. If the threshold is too high, effective balancing cannot be carried out. We propose an adaptive model under the adjustment of a central supervisor to update the threshold on a periodic basis. We have developed a prototype dynamic load balancer on a 32-node Intel iPSC/2 hypercube computer to evaluate the performance of the proposed methods. The benchmark experiments have verified the effectiveness of the load balancing scheme by achieving speedup of the parallel program execution.

Figure 1: Imbalance in creating processes among 4 processors in a multicomputer system.

Distributed Load Balancing under Supervision

A multicomputer is presented 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 \( G_i \) is defined as \( G_i = \{N_j | d_{ij} = 1\} \) for each node \( N_i \).

Figure 2: A generic model of the adaptive load balancing.

A generic model of the adaptive load balancing is shown in Fig.2, which will be used to describe different heuristic methods defined in the next section. The central supervisor is connected to all nodes by an information updating line, where the load index \( I_i \) of each node \( N_i \) is passed to the supervisor and the system load distribution \( L = \{I_i | 0 \leq i < n\} \) is broadcast to each node periodically. The invocation
of load balancing activities are totally distributed to the nodes. Each node \( N_i \) has two migration ports: the input migration port \( I_i \) and the output migration port \( O_i \). They are connected by the process migration network, which is defined by the load balancing method. Processes created at each node can be either executed locally or migrated to remote nodes for execution.

The supervisor periodically collects local load indices \( l_i \)'s and broadcasts the system load distribution \( L \) from/to all nodes. The load distribution \( L_i \) at time \( t \) is described by a mean value \( \bar{L}_i = \sum_{j} l_{ij} \) and a variance \( \sigma(L_i) = \sum_{j} (l_{ij} - \bar{L}_i)^2 \).

Let \( L_i \) and \( L_{i+1} \) \((i \geq 0)\) be the load distributions at two adjacent update times \( t_i \) and \( t_{i+1} \) respectively. The time window is defined as \( W_{ti} = t_{i+1} - t_i \). Let \( r \) be the load variation factor defined by: \( r = \frac{\max(0, l_i) - \min(0, l_{i+1})}{\max(0, l_i) + \min(0, l_{i+1})} \). The parameter \( r \) indicates the incremental change in two successive system load distributions. Assume the initial time windows \( W_{ti} = W_{t1} \). Consider two adjacent update times \( t_i \) and \( t_{i+1} \) and choose \( 0 < k_1 < k_2 < 1 \), the time window \( W_{t_{i+1}} \) is computed from the earlier window \( W_{ti} \) recursively as follows:

\[
W_{t_{i+1}} = \begin{cases} 
(1 - r) \cdot W_{ti} & \text{if } k_1 \leq r \leq k_2 \\
(1 - k_1) \cdot W_{ti} & \text{if } r > k_2 \\
(1 - k_1) \cdot W_{ti} & \text{if } r < k_1 \\
W_{ti} & \text{if } W_{ti} < k_2 \cdot W_{ti}
\end{cases} 
\]

Based on Eq. (1), the time window will become longer as the system enters 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 initialization period.

Figure 3: A model for the load balancer at each distributed processor node.

The load balancing at each node \( N_i \) is represented by the model shown in Fig.3. Ready processes at each node can be put into two queues: the ready queue \( R_i \) and the migration queue \( M_i \), respectively. The load index \( l_i \) of \( N_i \), is determined by the summation of a function \( F \) of each process \( p_k \) in the ready queue. That is \( l_i = \sum_{k \in R_i} F(p_k) \), where \( F(p_k) \) reflects the CPU time and memory demand of \( p_k \), which can be determined by the implementation and application requirements. The threshold \( \delta_i \) of \( N_i \) is determined by the load balancing method. When a process \( p_k \) is ready to run at \( N_i \), the load index \( l_i \) is compared with the threshold \( \delta_i \) by a decision maker. If \( l_i \leq \delta_i \), the process is put into the ready queue and \( l_i \) is incremented by \( F(p_k) \); otherwise it is put into the migration queue. As entered into the ready queue, the process will be executed by the CPU with no further transferring. If a process is put into the migration queue, it will be migrated to a remote node for execution. Processes which enter node \( N_i \) externally through \( I_i \) are always put into the ready queue. This guarantees that each process will be migrated at most once. The CPU scheduling discipline is the first-come first-serve (FCFS).

### Heuristic Methods for Dynamic Load Balancing

We propose four sender-initiated dynamic load balancing methods invoked by distributed nodes, which use heuristics to decide the invocation of load balancing, the threshold calculation, and the destination of process migration. The only information used in the decision making is from the system load distribution \( L_i \), received most recently from the supervisor. There is no information exchange among distributed nodes. By considering four combinations of the following two attributes, we propose four different heuristic methods for process migration.

1. **Decision range:**
   The load redistribution can be restricted to those nodes in the neighborhood set \( G_i \) adjacent to each \( N_i \), or involve all the multiple nodes in the system. The threshold is updated by 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 only migrates processes 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.

After receiving system load distribution \( L_i \) from the supervisor, the node \( N_i \) resets its local threshold \( \delta_i \) to \( \left( 1 + \alpha \right) \cdot \frac{\sum_{j} l_{ij}}{d_i + d_{i+1}} \), where \( 0 \leq \alpha \leq 0.2 \) is a normalized constant.

The input migration port \( I_i \) is connected to the output port \( O_i \)'s and the output migration port \( O_i \) is connected to the input port \( I_j \)'s, such that \( N_j \in G_i \). An ordered candidate list \( C_i \) is used to 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 \( W_{ti} \), \( C_i \) is updated in a Round-Robin fashion.

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 based on the Round-Robin discipline. After receiving the load distribution $L_i$ from the host, the global threshold $\delta_i$ is set to the system average load among all the nodes. That is $\delta_i = [(1 + \alpha) \cdot \frac{\sum_{j=1}^{n} L_j}{n}]$ for the time window $W_t$. The input and output migration ports $I_i$ and $O_i$ are connected to all $O_j$'s and $I_j$'s respectively, for $j \neq i$. 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 is in the policy to select a destination node. At node $N_i$, there is a load table to store the load index of each node in $G_i$. LML 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 setup threshold and migration ports is the same as that in GRR. But the destination node is determined as the same way as that in LML. That is the node with a minimum load index in the global load table will be selected as the destination node.

Parallelizing Program Executions at Process Control Level

We exploit the parallelism of a user program at the process control level, where each process is an atomic execution unit. The creation and suspension of a process is controlled by two operating system directives: run and suspend.

<table>
<thead>
<tr>
<th>PCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process ID (PID)</td>
</tr>
<tr>
<td>Parent ID</td>
</tr>
<tr>
<td>Port ID</td>
</tr>
<tr>
<td>State</td>
</tr>
<tr>
<td>Argument Counter</td>
</tr>
<tr>
<td>Number of Arguments</td>
</tr>
<tr>
<td>Code Address</td>
</tr>
<tr>
<td>Argument Array</td>
</tr>
</tbody>
</table>

Figure 4: The structure of a process control block (PCB).

Each process may be in one of the following states: new, ready, running, waiting and halted. The process state transitions are manipulated by three queues: ready, suspend and migration queue. A process is represented by a process control block (PCB). The structure of a PCB is shown Fig.4. Each process is identified by its Process ID (PID,PNODE). The parent ID and port ID are used for returning results to the parent process. When a process is halted, the return value along with its parent ID is stored in a data structure called result. A result of a process is always returned to its PNODE by message passing. At PNODE, the parent process in the suspend queue can be identified. The actual return value is returned to the appropriate port of the parent process. The argument counter counts the number of returns waiting from the child processes. A process is ready when its argument counter becomes 0. We use the complete copying scheme for process migration, where a copy of each user code is kept at each processor node. It is the PCB of a process being migrated rather than the actual code of a process. This can reduce the communication overhead caused by process migration.

Run and suspend O/S directives were originally suggested by Chowkwanyun for concurrent LISP program execution on multicomputers [1]. We extended them to the Unix/C programming environment.

- Run

The run() operation creates a new process by assigning it a PCB. The current process is the parent process of this newly created process. The parameters of run() specify the code address, the port.id identifying the return port to the parent process, the number of the arguments and the actual argument data.

- Suspend

The suspend() operation suspends the current process to wait for some results from child processes. The parameters of suspend() specify the code address from where the execution will start when the process is resumed, how many arguments are expected to be returned from the child processes and the available argument data.

Using run and suspend directives, a user program can generates a process tree at run time as shown in Fig.1. If the tree is unbalanced, then child processes can be migrated to idle nodes for execution.

A Dynamic Load Balancer

In addition to the simulated results of the proposed methods [8], we have developed a prototype dynamic load balancer at a 32-node Intel iPSC/2 hypercube system to carry out the benchmark experiments. The load balancer is written in C with iPSC/2 message passing library calls. It consists of a host program and a node program. The host program is implemented at the frontend host machine with the Unix O/S. It serves as the central supervisor as shown in Fig.5. The node program is implemented at the multiple hypercube computer nodes on top of the NX/2 operating system. Each identical node program serves as a dynamic load balancer which is shown Fig.6.

Figure 5: The construction of the host program.
Performance Evaluation by Benchmark Experiments

The benchmark programs chosen have unpredictable runtime conditions. They are C programs with run and suspend O/S directives inserted. We have done experiments on several programs [9]. Due to page limits, we only report the experiment on two recursive functions tak and fibonacci, whose execution time are totally data dependent and unable to be estimated at the compile time. For example, tak(18,16,9) needs 11842 calls while tak(18,16,15) only needs 7 calls.

The parallelized benchmark program fibonacci fib(x) is shown below:

```c
int fib(x)
int x;
{
  void suspend();
  int plus();
  int run();
  int fib();
  if (x <= 2)
    return(x);
  else
    
    suspend(plus,0,2);
    run(fib,0,1,x-1);
    run(fib,1,i,x-2);
    return(NONE); /* dummy return */
}
```

When the fib(x) program is loaded into a node, the kernel starts its execution by calling run(fib,0,0,x). This will create the first process p0, and put it to run. If x is greater than 2, then process p0 is suspended, with the next execution function as plus(), port ID as 0, and waiting for 2 data arguments. Two child processes p1 and p2 are created with argument x-1 and x-2 respectively. When these two child processes halt, the results will become the new arguments of the next "plus" execution of P0. Process p0 will be ready when these two arguments are available. Each ready process can be either executed locally or be migrated to remote node for execution. Thus, the parallelism at process control level can be exploited.

We use the benchmark programs to calculate the summation of a set of recursive functions. The execution of each recursive function is distributed to one node in the multiprocessor. We intend to make data arguments cause unbalanced process creations at multiple nodes. We have carried out two groups of experiments:

- **Group 1:**
  - Tak: calculate \( \sum_{i=0}^{x-2} \text{tak}(18,16,15) + \text{tak}(18,16,9) \).
  - Let \( \text{tak}(18,16,9) \) which needs 11842 calls to be executed at node \( N_i \).
  - Let \( \text{tak}(18,16,15) \) which needs 7 calls to be executed at node \( N_i \) such that \( i \neq 1 \).
  - Fibonacci: calculate \( \sum_{i=0}^{x-2} \text{fib}(3) + \text{fib}(17) \).
  - Let \( \text{fib}(17) \) which needs 3193 calls to be executed at node \( N_i \).
  - Let \( \text{fib}(3) \) which needs 3 calls to be executed at node \( N_i \) such that \( i \neq 1 \).

- **Group 2:**
  - Tak: calculate \( \sum_{i=0}^{x-2} \text{tak}(18,16,j), 9 \leq j \leq 15 \).
  - Let \( \text{tak}(18,16,j) \) be executed at each node, such that \( j = \text{random}(9,15) \).
  - Fibonacci: calculate \( \sum_{i=0}^{x-2} \text{fib}(j), 1 \leq j \leq 17 \).
  - Let \( \text{fib}(j) \) be executed at each node, such that \( j = \text{random}(1,17) \).

Group 1 creates extremely unbalanced load distribution, while Group 2 generates random load distribution.

We evaluated the performance by the speedup of parallel execution of benchmark programs. The speedup \( S_p \) to execute a program is calculated by \( S_p = \frac{T_1}{T_p} \), where \( T_1 \) is the program execution time using 1 node, and \( T_p \) is the program execution time using \( n \) nodes. We show only the results from Group 1 experiments in Fig. 7 and Fig. 8, where the four heuristic methods are compared with NO-LB (no load balancing) and GRAD (gradient) method [4]. For the Group 2 experiments, relatively higher speedup can be obtained, and there are some speedup even for NO-LB.

![Figure 7: Speedup obtained from executing program tak](image)

The four heuristic methods all perform much better than the gradient methods. The differences between them are not very clear. There are two reasons: 1) the system size \( n \) used in the experiments is bounded by 32 (facility limits), 2) the interconnection network is restricted to point-to-point communication. For point-to-point interconnection networks, we expect that the local method will be better.
Dynamic load balancing is extremely important for AI applications running on multicomputers [7]. Because AI-oriented programs have unpredictable run-time behavior, the scalability is hard to achieve without dynamic load balancing. Our prototype implementation and benchmark experiments verify the effectiveness of the load-balancing methods proposed. The methods can be adapted by a distributed operation system on a multicomputer.

Conclusions

We have proposed an adaptive model with four heuristic methods for dynamic load balancing in a multicomputer system. These methods require less control overhead compared with previously proposed methods. This is accomplished by using a central supervisor to collect system load information and update load balancing threshold. Since the host supervisor is overlapped with distributed node executions, less overhead is introduced. The threshold used by those heuristics is adaptively determined using the most recently updated system state. We use the Round-Robin discipline on a candidate list of nodes and minimum load strategy to determine the destination of the load transferring, which can be used for different multicomputer environments. The performance of these methods is evaluated by parallel execution of benchmark programs under the control of a dynamic load balancer implemented at an IPSC/2 hypercube computer. Benchmark programs are parallelized at the process control level by using the run and suspend O/S directives. The speedup of the parallel program execution can be achieved by applying dynamic load balancing, even if the program is totally data dependent. Although the experiments are performed at a hypercube architecture, the proposed methods are suitable for various interconnection network topologies. One can make a choice among these four methods based on the multicomputer topology used. For very large systems, a hierarchical clustering method is suggested in [9], in which the proposed model can be used within each cluster.

References


