Chapter 7

Load Balancing and Termination Detection

- Load balancing is used to distribute computations fairly across processors in order to obtain the highest possible execution speed.
- Termination detection is detecting when a computation has been completed:
  - relatively easy to do when the process and computation structure is fixed,
  - becomes a significant issue when the computation is distributed.

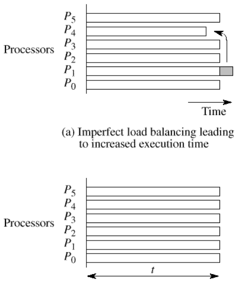
Load Balancing

So far:

- A problem was divided into a fixed number of processes where each process performs a known amount of work.
- Processors assumed to be equivalent, i.e., same type and same speed.

Load balancing - to obtain minimum execution time.

- most useful when the amount of work is not known prior to execution.
- helps to mitigate the effects of differences in processor speeds even when the amount of work is known in advance.



(a) Imperfect load balancing leading to increased execution time

(b) Perfect load balancing

Figure 7.1 Load balancing.

Static Load Balancing

Also known as mapping problem or scheduling problem. (Bokhari, 1981)

- Load balancing is attempted statically before the execution of any process.
- Substantial literature exists on the problem, mostly using the optimization techniques (estimating execution times and interdependencies in program)

Some Potential Load Balancing Techniques

- Round robin algorithm
- Randomized algorithms
- Recursive bisection:
  - recursively divides the problem into subproblems of equal computational effort while minimizing message passing.
- Simulated annealing
- Genetic algorithms

Bin packing: placing objects into boxes to reduce the number of boxes. Scheduling can be approached with bin packing algorithms (Coffman et al., 1978)

A Mapping Problem

Goal: to reduce the communication delays

Question: how do we map processes to processors?

Network Architecture: Processors/computers interconnected by a static link ICNW

Solution:

Communicating processors should be executed on processors with direct communication paths

- Computationally intractable - NP-complete.
- Therefore, often heuristics are used for the solution.

Fundamental flaws in static load balancing:

- time estimation is difficult without running the program first
  - scheduling parts of a program without using actual execution times is innately inaccurate.
- It could be difficult to incorporate *variable* comm. time delays in static load balancing.
  - Consider the problems with indeterminate number of steps to reach the solution.

Dynamic Load Balancing

Takes all related factors into account by making the division of load dependent upon the execution of the parts as they are being executed

- additional overhead during execution
- how a computation finally comes to an end - termination detection is a related significant problem.

The computation will be divided into *work* or *tasks* to be performed, and processes perform these tasks.

- a single process operates upon tasks
- there needs to be at least as many tasks as processors
- Objective: to keep the processors busy.

Two types of dynamic load balancing:

- centralized: tasks are handed out from a centralized location: master-slave structure.
- decentralized: tasks are passed between arbitrary processes: workers structure.

Centralized Dynamic Load Balancing

- Master process(or) holds the collection of tasks to be performed and sends them to slave processes when necessary.
- Completing slave processes requests another task from the master process.

- Similar to “work pool approach”
- Sometimes called “replicated worker” or “processor farm” since all the slaves are the same.

Disadv: master process can only issue one task at a time, and after the initial tasks have been sent, it can only respond to requests for new tasks one at a time - potential bottleneck!

Work Pool Approach

Can be readily applied to

- simple divide-and-conquer problems.
- to problems in which the tasks are quite different and of different sizes.
- when the number of tasks may change during execution: e.g., search algorithms.

As a rule of thumb, hand out the larger/most complex tasks first.

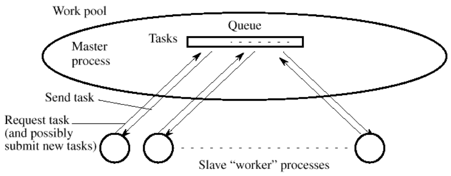


Figure 7.2 Centralized work pool.

Termination.

Stopping the computation when the solution has been reached.

Termination occurs when both conds are satisfied:

- the task queue is empty
- every process has made a request for another task without any new tasks being generated.

In some applications, a slave may detect the termination condition by some local termination condition, e.g., finding an item in a search alg.

In some applications, each slave process must reach a specific local termination condition, e.g., convergence on its local solutions.

Decentralized Dynamic Load Balancing

- Centralized approach would be satisfactory when there are few slaves and the tasks are computationally intensive.
- When finer-grain tasks and many slaves, it may be more appropriate to distribute the work pool into more than one site.

Several levels of decomposition is possible.

Slaves can hold a portion of the work pool and solve for this portion.

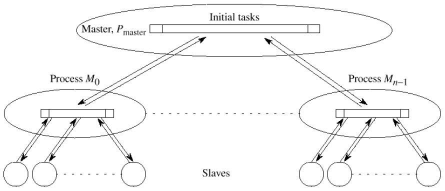


Figure 7.3 A distributed work pool.

Fully Distributed Work Pool

The tasks could be transferred by

1. The receiver-initiated method - a process requests tasks from other processes it selects

- works well at high system load.

2. The sender-initiated method - a process sends tasks to other processes it selects

- works well under light overall system loads.

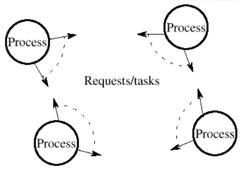


Figure 7.4 Decentralized work pool.

Load balancing with receiver-initiated method:

- Organize processes as a ring with a process requesting tasks from its nearest neighbors.
- In a hypercube, . . .

Process Selection.

Without the constraints of a specific network, all processors are equal candidates and processes could select any other process.

Local implementations:

- round robin algorithm
- random polling algorithm

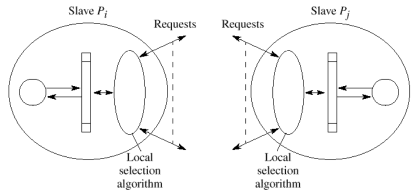


Figure 7.5 Decentralized selection algorithm requesting tasks between slaves.

Load Balancing Using a Line Structure (Wilson, 1995)

Basic idea: create a queue of tasks with individual processors accessing locations in the queue.

- High priority or larger tasks could be placed in the queue first.

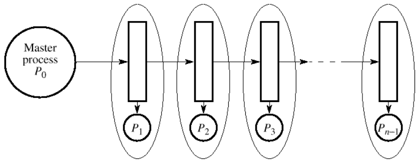


Figure 7.6 Load balancing using a pipeline structure.

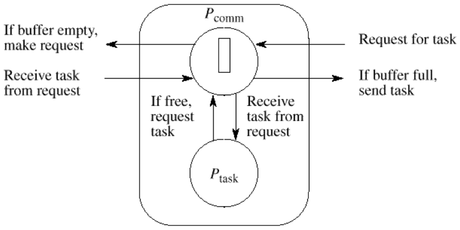

P_{comm}
If buffer empty, make request
Request for task
Receive task from request
If buffer full, send task
If free, request task
Receive task from request
P_{task}

Figure 7.7 Using a communication process in line load balancing.

Two processes running on each processor:

- for left and right communication
- for the current task

Master Process (P_0):

```
for (i=0; i<no_tasks; i++) {
   recv(P1, request_tag);
   send(&task, Pi, task_tag);
}
recv(P1, request_tag);
send(&empty, Pi, task_tag);
```

Process P_i (1<i<n) :

```
if (buffer == empty) {
   send(Pi-1, request_tag);
   recv(buffer,Pi-1,task_tag);
}
if (buffer==full)&&(!busy)){
   task = buffer;
   buffer = empty;
   busy = TRUE;
}
nrecv(Pi+1,request_tag,request);
if (request && (buffer==full)){
   send(&buffer, Pi+1);
   buffer = empty;
}
if (busy) {
   Do some work on task;
   busy = FALSE;
}
```

Nonblocking Receive Routines:

Consider **nrecv(P$_{i+1}$,request_tag,request);**

`request` is set to TRUE if a message has been received.

pvm_nrecv() returns a zero value if no message has been received.

pvm_probe() used to check whether a message has been received without actually reading the message. Subsequently a normal **recv()** routine is needed to accept and unpack the message.

MPI_Irecv() posts a request message and returns immediately. But it returns a request handle, which is used in subsequent completion routines (**MPI_Wait()** and **MPI_Test()**) to wait for the message or to establish whether the message has actually been received at that point.

Other Structures (A Tree like structure by Wilson, 1995)

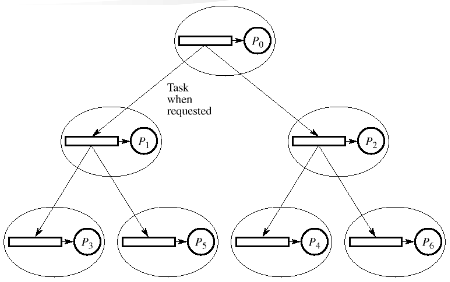


Figure 7.8 Load balancing using a tree.