Static Load Balancing

Before the execution of any process, the computation is distributed.

Some potential static load-balancing techniques:
- **Round robin algorithm** — passes out tasks in sequential order of processes coming back. Older processes get tasks if tasks are available.
- **Randomized algorithm** — selects processes at random to take tasks.
- **Recursive algorithm** — recursively divides the problem into subproblems of equal computational effort while minimizing message passing.
- **Simulated annealing** — another optimization technique, described in Chapter 12.

Figure 7.1 could also be viewed as a form of bin packing (that is, placing objects into boxes to reduce the number of boxes).

In general, computationally intractable problems, so-called \( \text{NP} \)-complete.

\( \text{NP} \) stands for “nondeterministic polynomial” and means there is probably no polynomial-time algorithm for solving the problem. Hence, often heuristics are used to select processors for processes.

Several fundamental flaws with static load balancing even if a mathematical solution exists:
- Very difficult to estimate accurately the execution times of various parts of a program without actually executing the parts.
- Communication delays that vary under different circumstances.
- Some problems have an indeterminate number of steps to reach their solution.

Load Balancing and Termination Detection

**Load balancing** — used to distribute computations fairly across processors in order to obtain the highest possible execution speed.

**Termination detection** — detecting when a computation has been completed. More difficult when the computation is distributed.

In general, computationally intractable problems, so-called \( \text{NP} \)-complete.

\( \text{NP} \) stands for “nondeterministic polynomial” and means there is probably no polynomial-time algorithm for solving the problem. Hence, often heuristics are used to select processors for processes.

Some problems have an indeterminate number of steps to reach their solution.
Dynamic Load Balancing

Processes and Processors

Does not incur an overhead during execution, but it is much more effective than the execution of the entire program.

All execution phases are taken into account by making the decision of load assignment.
Termination

In this case, the master must receive termination messages from all the slaves.

In some applications, each slave process must reach a specific local termination condition.

In the case of the slave process reaching the end, the termination condition must be reached.

In centralized load balancing, the master holds the collection of tasks to be performed.

Terms used: work pool, replicated worker, processor farm.

Figure 7.2

Centralized work pool.

Slave "worker" processes

Master process(or) holds the collection of tasks to be performed.

Tasks are sent to the slave processes when a slave process completes one task. If no tasks are sent to the slave processes, when a slave process completes one task, the master must receive termination messages from all the slaves.

In some applications, a slave may detect the condition of termination by some local termination condition. In that case, the slave must notify the master of the termination of the task.

In some applications, each slave process must reach a specific local termination condition.

In this case, the master must receive termination messages from all the slaves.

Centralized Dynamic Load Balancing

Tasks are sent to the slave processes when a slave process completes one task. If no tasks are sent to the slave processes, when a slave process completes one task, the master must receive termination messages from all the slaves.
Figure 7.3: A distributed work pool.

Processes to execute tasks from each other

Figure 7.4: Fully distributed work pool.

Decentralized Dynamic Load Balancing

Distributed Work Pool
Task Transfer Mechanisms

**Task Transfer Mechanisms**

**Receiver-Initiated Method**
A process requests tasks from other processes it selects. Typically, a process would request tasks from other processes when it has few or no tasks to perform. This method has been shown to work well at high system load.

**Sender-Initiated Method**
A process sends tasks to other processes it selects. Typically, in this method, a process with a heavy load passes out some of its tasks to others that are willing to accept them. This method has been shown to work well for light overall system loads. Another option is to have a mixture of both methods.

Unfortunately, it can be expensive to determine process loads. In very heavy system loads, load balancing can also be difficult to achieve because of the lack of available processes.

**Process Selection**

**Round robin algorithm**
- Process $P_i$ requests tasks from process $P_x$, where $x$ is given by a counter that is incremented after each request, using modulo $n$ arithmetic (where $n$ processes), excluding $x = i$.

**Random polling algorithm**
- Process $P_i$ requests tasks from process $P_x$, where $x$ is a number that is selected randomly between 0 and $n-1$ (excluding $i$).

In very heavy system loads, load balancing can also be difficult to achieve because of the lack of available processes. An alternative is to have a mixture of both methods.

**Another option is to have a mixture of both methods.**
Load Balancing Using a Line Structure

Figure 7.6

Load balancing using a pipeline structure.

The master process (P₀ in Figure 7.6) feeds the queue with tasks at one end, and the tasks are shifted down the queue. When a "worker" process, Pᵢ (1 ≤ i < n), detects a task at its input from the queue and the process is idle, it takes the task from the queue. Then the tasks to the left shuffle down the queue so that the space held by the task is filled. A new task is inserted into the left side of the queue. Eventually, all processes will have a task and the queue is filled with new tasks. High-priority or larger tasks could be placed in the queue first.

Figure 7.7

Using a communication process in line balancing.

If buffer empty, make request
Receive task from request
If free, request task
Receive task from request
If buffer full, send task
Request for task

Shifting Actions

Shifting actions could be orchestrated by using messages between adjacent processes.
Master process ($P_0$)

for ($i = 0; i < \text{no\_tasks}; i++) {
    recv($P_1$, request\_tag); /* request for task */
    send(&task, $P_i$, task\_tag); /* send tasks into queue */
}
recv($P_1$, request\_tag); /* request for task */
send(&empty, $P_i$, task\_tag); /* end of tasks */

Process $P_i$ ($1 < i < n$)

if ($\text{buffer} == \text{empty}$) {
    send($P_{i-1}$, request\_tag); /* request new task */
    recv(&buffer, $P_{i-1}$, task\_tag); /* task from left proc */
}
if ($\left(\text{buffer} == \text{full}\right) \land \left(\neg \text{busy}\right)$) { /* get next task */
    task = buffer; /* get task*/
    buffer = empty; /* set buffer empty */
    busy = true; /* set process busy */
}
recv($P_{i+1}$, request\_tag, request); /* check message from right */
if (request && ($\text{buffer} == \text{full}$)) {
    send(&buffer, $P_{i+1}$); /* shift task forward */
    buffer = empty;
}
if (busy) { /* continue on current task */
    \text{Do some work on task. If task finished, set busy to false.}
}

In this code, a combined $\text{send}$/$\text{recv}$ might be applied if available rather than a $\text{send}$/$\text{recv}$ pair.

A nonblocking $\text{nrecv}$ is necessary to check for a request being received from the right.

In our pseudocode, we have simply added the parameter $\text{request}$, which is set to true if a message has been received.

Nonblocking Receive Routines

In PVM, a nonblocking receive event is needed to check and unpack the message without actually receiving the message. A probe function (PVM$\text{nrecv}$) could be used to check whether a message has been received.

In MPI, a nonblocking receive (MPI$\text{Irecv}$) returns a request handle, which is used in subsequent completion routines to wait for the message or to establish whether the message has actually been received (MPI$\text{Wait}$ and MPI$\text{Test}$, respectively).

Nonblocking MPI$\text{Irecv}$ posts a request for message and events immediately.

Nonblocking receive (MPI$\text{Irecv}$) returns a handle that is zero if no message has been received.
Elaboration of pipeline approach to a tree.

Tasks are passed from a node into one of the two nodes below it when a node buffer becomes empty.

Figure 7.8 Load balancing using a tree.
Using Acknowledgment Messages

Each process is in one of two states:

- **Active**: A process is active if it has not yet terminated. An active process can receive a task and can send a request for acknowledgment.
- **Inactive**: A process is inactive if it has terminated and has not yet sent a request for acknowledgment.

When a process changes from active to inactive, it sends an acknowledgment message to its parent. The parent process can then change to inactive if it has received all acknowledgments for tasks it has sent out.

### Ring Termination Algorithms

**Single-pass Ring Termination Algorithm**

1. When process 0 has terminated, it generates a token that is passed to process 1.
2. When process 0 receives the token and has already terminated, it passes the token on to process 1.
3. When process 0 receives a token, it knows that all processes in the ring have terminated. A message can then be sent to all processes informing them of global termination, if necessary.

The algorithm assumes that each process can be reached after reaching its local termination condition.

### Using Acknowledgment Messages

- When the last process becomes idle, the computation can terminate.
- If a process receives all acknowledgment messages for tasks it has sent out, it becomes inactive.
- If a process has received all acknowledgment messages for tasks it has received, it becomes inactive.
- If a process has fulfilled all local termination conditions (all tasks are completed), it becomes inactive.

If any process sends an acknowledgment message to a process when it is ready to become inactive, it must send an acknowledgment message to the process where the token is currently located.

When the token reaches process 0, it indicates that all processes have terminated. A message can then be sent to all processes informing them of global termination, if necessary.
Dual-Pass Ring Termination Algorithm

1. When $P_0$ receives a black token, it passes on a white token. If it receives a white token, it becomes a black process. When $P_{n-1}$ passes a token to $P_0$, $P_0$ becomes the central point for global termination.

2. When $P_i$, $i > 0$, passes a task to $P_{i-1}$ (that is, before this process in the ring), it becomes a black process. If after a task is passed on the token, the process color is reversed, a white token will be generated. If the ring is not empty, the token must be recirculated around the ring a second time.

3. The algorithm is as follows:

   - $P_0$ becomes white when it has terminated and generates a white token.
   - Each process is either white or black.
   - A black process becomes white when it receives a black token.
   - A white process becomes black when it receives a black token.

The algorithm is as follows:

1. $P_0$ receives a black token and becomes white. It generates a white token.
2. The token passes through the ring from one process to the next, and each process may change the color of the token. If $P_{n-1}$ receives a token, it becomes a black process.
3. $P_{n-1}$ passes the token to $P_0$. If $P_0$ receives a black token, it passes on a white token; if it receives a white token, all processes have terminated.

Notice that in both ring algorithms, $P_0$ becomes the central point for global termination. Also, it is assumed that an acknowledge signal is generated to each request.
Tree Algorithm

The local actions described in Figure 7.11 can be applied to various interconnection structures, notably a tree structure. To indicate that processes up to that point have terminated, a significant disadvantage of the fixed energy method is that dividing the energy will be of the nature of reinitiating a process.

When all the energy is returned to the root and the root becomes idle, all processes must terminate. When a process becomes idle, it passes the energy it holds back before requesting a new process to free up the energy. Similarly, if the process receive requests for tasks, the energy is divided among the processes. When a process becomes idle, it passes the energy it holds back to the process giving it the original task. This energy could be passed directly back to the master process or to the process giving it the original task.

A significant disadvantage of the fixed energy method is that dividing the energy will be of the nature of reinitiating a process. When all the energy is returned to the root and the root becomes idle, all processes must terminate. When a process becomes idle, it passes the energy it holds back before requesting a new process to free up the energy. Similarly, if the process receive requests for tasks, the energy is divided among the processes. When a process becomes idle, it passes the energy it holds back to the process giving it the original task. This energy could be passed directly back to the master process or to the process giving it the original task.

A significant disadvantage of the fixed energy method is that dividing the energy will be of the nature of reinitiating a process. When all the energy is returned to the root and the root becomes idle, all processes must terminate. When a process becomes idle, it passes the energy it holds back before requesting a new process to free up the energy. Similarly, if the process receive requests for tasks, the energy is divided among the processes. When a process becomes idle, it passes the energy it holds back to the process giving it the original task. This energy could be passed directly back to the master process or to the process giving it the original task.
The shortest path problem can be stated as follows:

Given a set of interconnected nodes where the links between the nodes are marked with "weights," find the path from one specific node to another specific node that has the smallest accumulated weights.

The interconnected nodes can be described by a graph. In graph terminology, the nodes are called vertices, and the links are called edges.

If the edges have implied directions (that is, an edge can only be traversed in one direction, the graph is a directed graph.

The graph itself could be used to find the solution to many different problems; for example:

1. The shortest distance between two towns or other points on a map, where the weights represent distance.
2. The quickest route to travel, where the weights represent time (the quickest route may not be the shortest route if different modes of travel are available; for example, flying to certain towns).
3. The least expensive way to travel by air, where the weights represent the cost of the flights between cities (the vertices represent the cities).
4. The best route through a network to minimize message delay (the vertices represent the computers, and the weights represent the delay between two computers).
5. The best route through a network to minimize certain network criteria (for example, bandwidth).
6. The best way to climb a mountain given a certain map with contours.

"The best way to climb a mountain" will be used as an example.

1. The node (or mountain) at which we start (the source).
2. The node (or mountain) at which we finish (the destination).
3. The graph that represents the map of the terrain.
4. The weight on each edge represents the height of the mountain.

The graph could be used to find the solution to many different problems; for example:

1. The shortest distance between two points on a map.
2. The quickest route to travel between two points.
3. The least expensive way to travel between two points.
4. The best way to climb a mountain.
5. The best route through a network to minimize message delay.
6. The best manufacturing system, where the weights represent hours of work.

Given a path from the source node to the destination node, the weight of the path represents the total height climbed.

The shortest path problem is important in many applications, such as in network routing, where the goal is to find the path with the minimum delay for data transmission.

Example: The best way to climb a mountain.
Weights in graph indicate the amount of effort that would be expended in traversing the route between two connected camp sites. The effort in one direction may be different from the effort in the opposite direction (downhill instead of uphill!). (directed graph)

Graph Representation

Two basic ways that a graph can be represented in a program:

1. Adjacency matrix — a two-dimensional array, \( a[i][j] \), in which \( a[i][j] \) holds the weight associated with the edge between vertex \( i \) and vertex \( j \) if one exists.

2. Adjacency list — for each vertex, a list of vertices directly connected to the vertex by edges and the corresponding weights associated with the edges.

The adjacency matrix is used for dense graphs, where there are many edges between vertices. The adjacency list is used for sparse graphs, where there are fewer edges. The difference is based upon space (storage) requirements. Adjacency matrix has \( O(n^2) \) space requirement and adjacency list has an \( O(nv) \) space requirement, where there are \( v \) edges from each vertex and \( n \) vertices in the graph.

Accessing the adjacency list is slower than accessing the adjacency matrix, as it requires the linked list to be traversed sequentially, which potentially requires \( v \) steps.
Searching a Graph

TWO WELL-KNOWN SINGLE-SOURCE SHORTEST-PATH ALGORITHMS: 

- Moore’s single-source shortest-paths algorithm (Moore, 1959)
- Dijkstra’s single-source shortest-path algorithm (Dijkstra, 1959)

The weights must all be positive values for the algorithm to work. (Other algorithms exist although they may do more work.

Dijkstra’s algorithm is chosen because it is more amenable to parallel implementation

Which are similar:

- Moore’s algorithm is chosen because it is more amenable to parallel implementation

Two well-known single-source shortest-path algorithms:

- Moore’s single-source shortest-paths algorithm (Moore, 1959)
- Dijkstra’s single-source shortest-path algorithm (Dijkstra, 1959)
Starting with the source vertex, the basic algorithm implemented when vertex $i$ is being considered as follows.

Find the distance to vertex $j$ through vertex $i$ and compare with the current minimum distance to vertex $j$. Change the minimum distance if the distance through vertex $i$ is shorter.

In mathematical notation, if $d_i$ is the current minimum distance from the source vertex to vertex $i$ and $w_{i,j}$ is the weight of the edge from vertex $i$ to vertex $j$, we have

$$
d_j = \min(d_j, d_i + w_{i,j})$$

(Each edge $(i,j)$ is of the form $(i, j, w_{i,j})$.)

The current shortest distance from the source vertex to vertex $j$ will be stored in the array $d_j$.

Suppose there are $n$ vertices, and vertex 0 is the source vertex.

Initially, none of these distances will be known and the array elements are initialized to infinity.

At this stage of these distances will be known and the array elements are initialized to

$$d_{j} = \infty \text{ for } 0 \leq j < n .$$

The current shortest distance from the source vertex to vertex $j$ will be stored in the array.

Suppose there are $n$ vertices and vertex 0 is the source vertices.

Another criterion is needed to hold the current shortest distance from the source vertex to

Initially, only the source vertex is in the queue.

Vertices are considered only when they are in the queue.

A first-in-first-out vertex queue is created and holds a list of vertices to examine.

**Data Structures and Code**

**Moore's Algorithm**

Starting with the source vertex, the basic algorithm implemented when vertex $i$ is being...
To see how this algorithm proceeds from the source vertex, let us follow the steps using our mountain climbing graph as the example.

The initial values of the two key data structures are:

<table>
<thead>
<tr>
<th>vertex</th>
<th>Current minimum distances</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>05</td>
</tr>
<tr>
<td>D</td>
<td>06</td>
</tr>
<tr>
<td>E</td>
<td>110</td>
</tr>
<tr>
<td>F</td>
<td>18</td>
</tr>
</tbody>
</table>

After examining A to B:

- B: 10

After examining B to F, E, D, and C:

- E: 05
- D: 06
- C: 05

After examining C to D: No change.

After examining E (again) to F:

- F: 010

After examining F: No change.

There are no more vertices to consider.

We have the minimum distance from vertex A to each of the other vertices, including the destination vertex, F.

Usually, the actual path is also required in addition to the distance. Then the path needs to be stored as the distances are recorded. The path in our case is A → B → D → E → F.
Sequential Code

The specific details of maintaining the vertex queue are omitted. Let
next_vertex() return the next vertex from the vertex queue or
no_vertex if none.

We will assume that an adjacency matrix is used, named
w[i][j], which is accessed sequen-
tially to find the next edge.
The sequential code could then be of the form

```c
while ((i = next_vertex()) != no_vertex) /* while a vertex */
    for (j = 1; j < n; j++) /* get next edge */
        if (w[i][j] != infinity) { /* if an edge */
            newdist_j = dist[i] + w[i][j];
            if (newdist_j < dist[j]) {
                dist[j] = newdist_j;
                append_queue(j); /* vertex to queue if not there */
            }
        } /* no more vertices to consider */
```

Parallel Implementations

Centralized Work Pool

Centralized work pool holds the vertex queue, vertex_queue[ ] as tasks.
Each slave takes vertices from the vertex queue and returns new vertices.

Since the edge list of maintaining the vertex queue are omitted, let
next_vertex() return the next vertex from the vertex queue and returns new vertices.

The edge list of maintaining the vertex queue is used, named w[i][j], which is accessed sequentially.

Sequential Code

```c
while (vertex_queue() != empty) {
    recv(PANY, source = Pi); /* request task from slave */
    v = get_vertex_queue();
    send(&v, Pi); /* send next vertex and */
    send(&dist, &n, Pi); /* current dist array */.
    recv(&j, &dist[j], PANY, source = Pi); /* new distance */
    append_queue(j, dist[j]); /* append vertex to queue */
    /* and update distance array */
}
```
A convenient approach is to assign slave process \( i \) to search around vertex \( i \) only and for it to have the vertex queue entry for vertex \( i \) if this exists in the queue. The array \( \text{dist}[\cdot] \) will also be distributed among the processes so that process \( i \) maintains the current minimum distance to vertex \( i \). Process \( i \) also stores an adjacency matrix/list for vertex \( i \), for the purpose of identifying the edges from vertex \( i \).

**Search Algorithm**

The search will be activated by a coordinating process loading the source vertex into the appropriate process. In our case, vertex \( A \) is the first vertex to search. The process assigned to vertex \( A \) is activated. This process will immediately begin searching around its vertex to find distances to connected vertices. The distance to process \( j \) will be sent to process \( j \) for it to compare with its currently stored value and replace if the currently stored value is larger. In this fashion, all minimum distances will be updated during the search. If the contents of \( d[\cdot] \) change, process \( i \) will be reactivated to search again.

**Decentralized Work Pool**
A code segment for the slave processes might take the form:

```c
Slave (process \(i\))
recv(newdist, PANY);
if (newdist < dist) 
  dist = newdist; vertex_queue = TRUE; /* add to queue */
else vertex_queue = FALSE;
if (vertex_queue == TRUE) /* start searching around vertex */
  for (j = 1; j < n; j++) /* get next edge */
    if (w[j] != infinity) 
      d = dist + w[j]; send(&d, Pj); /* send distance to proc j */
```

This could certainly be simplified to:

```c
Slave (process \(i\))
recv(newdist, PANY);
if (newdist < dist) 
  dist = newdist; /* start searching around vertex */
for (j = 1; j < n; j++) /* get next edge */
  if (w[j] != infinity) 
    d = dist + w[j]; send(&d, Pj); /* send distance to proc j */
```

A mechanism is necessary to repeat the actions and terminate when all processes are idle.

The mechanism must cope with messages in transit. The simplest solution is to use synchronous message passing, in which a process cannot proceed until the destination has received the message.

Note that a process is only active after its vertex is placed on the queue, and it is possible for many processes to be inactive, leading to an inefficient solution.

The method is also impractical for a large graph if one vertex is allocated to each processor. In that case, a group of vertices could be allocated to each processor.

The mechanism is also unpractical for a large graph, since it requires a large number of processors.

PROBLEMS

7-1. One approach for assigning processes to processors is to make the assignment random using a random number generator. Investigate this technique by applying it to a parallel program that adds together a sequence of numbers.

7-2. Write a parallel program that will implement the load-balancing technique using a pipeline structure described in Section 7.2.3 for any arbitrary set of independent arithmetic tasks.

7-3. The traveling salesperson problem is a classical computer science problem (though it might also be regarded as a real life problem). Starting at one city, the objective is to visit each of \(n\) cities exactly once and return to the first city on a route that minimizes the distance traveled. The \(n\) cities can be regarded as variously connected. The connections can be described by a weighted graph. Write a parallel program to solve the traveling salesman problem with real data obtained from a map to include 25 major cities.

7-4. Implement Moore's algorithm using the load-balancing line structure described in Section 7.2.3.

7-5. As noted in the text, the decentralized work pool approach described in Section 7.4 for searching a graph is inefficient in that processes are only active after their vertex is placed on the queue, and it is possible for processes with the same vertex to accept the message.

The mechanism must cope with messages in transit. A mechanism is necessary to repeat the actions and terminate when all processes are idle.

The mechanism is also unpractical for a large graph, since it requires a large number of processors.

7-6. Write a loading-balancing program using Moore's algorithm and a load-balancing program using Dijkstra's algorithm for searching a graph. Compare the performance of each algorithm and make conclusions.
7-7. Single-source shortest-path algorithms can be used to find the shortest route for messages in a multicomputer interconnection network, such as a mesh or hypercube network or any interconnection network one would like to devise. Write a parallel program that will find all the shortest routes through a \( d \)-dimensional hypercube, where \( d \) is input.

7-8. Modify the program in Problem 7-7 to handle an incomplete hypercube. An incomplete hypercube is one with one or more links removed. One form of incomplete hypercube consists of two interconnected complete hypercubes of size \( 2^n \) and \( 2^k \) (\( 1 \leq k \leq n \)). More details can be found in Tzeng and Chen (1994).

7-9. You have been commissioned to develop a challenging maze to be constructed at a stately home. The maze is to be laid out on a grid such as shown in Figure 7.19. Develop a parallel program that will find the positions of the hedges that result in the longest time in the maze if one uses the maze algorithm: "Keep to the path where there is a hedge or wall on the left" as is illustrated in Figure 7.19, which is guaranteed to find the exit eventually (Berman and Paul, 1997).

7-10. A building has a number of interconnected rooms with a pot of gold in one, as illustrated in Figure 7.20. Draw a graph describing the plan of rooms where each vertex is a room. Doors connecting rooms are shown as edges. Circle the position of the pot of gold. Notice that edges are bidirectional, and cycles may exist in the graph.