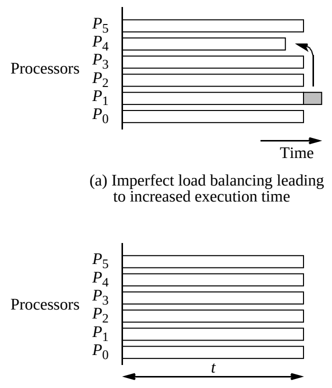
Load Balancing and Termination Detection

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 computaion is distributed.



(a) Imperfect load balancing leading to increased execution time

(b) Perfect load balancing

Figure 7.1 Load balancing.

Static Load Balancing

Before the execution of any process

Some potential static load-balancing techniques:

- *Round robin algorithm* — passes out tasks in sequential order of processes coming back to the first when all processes have been given a task
- *Randomized algorithms* — selects processes at random to take tasks
- *Recursive bisection* — recursively divides the problem into subproblems of equal computational effort while minimizing message passing
- *Simulated annealing* — an optimization technique
- *Genetic algorithm* — 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 problem, so-called *NP*-complete.

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.

Dynamic Load Balancing

During the execution of the processes.

All previous factors are taken into account by making the division of load dependent upon the execution of the parts as they are being executed.

Does incur an additional overhead during execution, but it is much more effective than static load balancing

Processes and Processors

Processes are mapped onto processors.

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

With this terminology, a single process operates upon tasks.

There needs to be at least as many tasks as processors and preferably many more tasks than processors.

Since our objective is to keep the processors busy, we are interested in the activity of the processors.

However, we often map a single process onto each processor, so we will use the terms *process* and *processor* somewhat interchangeably.

Dynamic Load Balancing

Tasks are allocated to processors during the execution of the program.

Dynamic load balancing can be classified as one of the following:

- Centralized
- Decentralized

Centralized dynamic load balancing

Tasks are handed out from a centralized location.
A clear master-slave structure exists.

Decentralized dynamic load balancing

Tasks are passed between arbitrary processes.

A collection of worker processes operate upon the problem and interact among themselves, finally reporting to a single process.

A worker process may receive tasks from other worker processes and may send tasks to other worker processes (to complete or pass on at their discretion).

Centralized Dynamic Load Balancing

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, it requests another task from the master process.

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

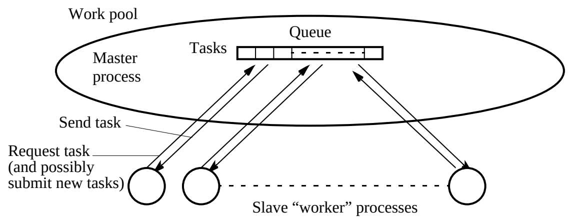


Figure 7.2 Centralized work pool.

Termination

Stopping the computation when the solution has been reached.

For a computation in which tasks are taken from a task queue, the computation terminates when both of the following are satisfied:

- The task queue is empty
- Every process has made a request for another task without any new tasks being generated

Notice that it is ***not* sufficient** to terminate when the task queue is empty if one or more processes are still running because a running process may provide new tasks for the task queue.

(Those problems that do not generate new tasks, such as the Mandelbrot calculation, would terminate when the task queue is empty and all slaves have finished.)

In some applications, a slave may detect the program termination condition by some local termination condition, such as finding the item in a search algorithm.

In that case, the slave process would send a termination message to the master. Then the master would close down all the other slave processes.

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

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

Decentralized Dynamic Load Balancing

Distributed Work Pool

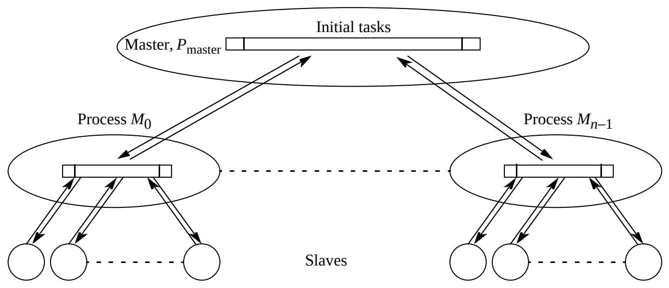


Figure 7.3 A distributed work pool.

Fully Distributed Work Pool

Processes to execute tasks from each other

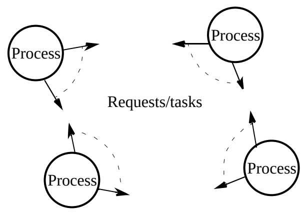


Figure 7.4 Decentralized work pool.

Task Transfer Mechanisms

Receiver-Initiated Method

Aprocess 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.

Method has been shown to work well at high system load.

Sender-Initiated Method

Aprocess 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.

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

Algorithms for selecting a process:

R*ound 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 (n processes), excluding $x = i$.
.

R*andom 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).

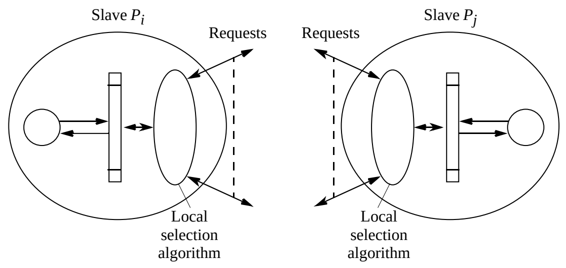


Figure 7.5 Decentralized selection algorithm requesting tasks between slaves.

Load Balancing Using a Line Structure

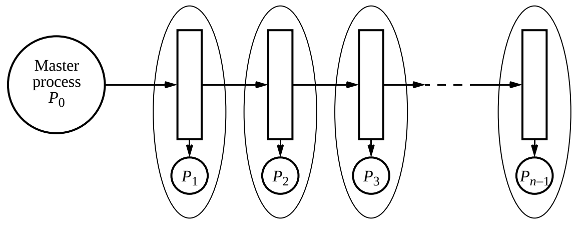


Figure 7.6 Load balancing using a pipeline structure.

The master process (P_0 in Figure 7.6) feeds the queue with tasks at one end, and the tasks are shifted down the queue.

When a "worker" process, P_i ($1 \leq i < n$), 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 end 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.

Shifting Actions

could be orchestrated by using messages between adjacent processes.

Perhaps the most elegant method is to have two processes running on each processor:

- For left and right communication
- For the current task

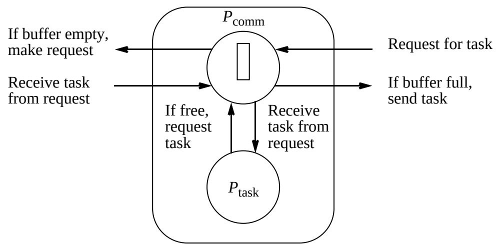

P_{comm}
If buffer empty, make request
Request for task
Receive task from request
If buffer full, send task
If free, request task
Receive task from request
P_{task}

Figure 7.7 Using a communication process in line load balancing.

Code Using Time Sharing Between Communication and Computation

Master process (P_0)

```
for (i = 0; i < no_tasks; i++) {
   recv(P1, request_tag);              /* request for task */
   send(&task, Pi, task_tag);          /* send tasks into queue */
}
recv(P1, request_tag);                 /* request for task */
send(&empty, Pi, task_tag);            /* end of tasks */
```

Process P_i ($1 < i < n$)

```
if (buffer == empty) {
   send(Pi-1, request_tag);            /* request new task */
   recv(&buffer, Pi-1, task_tag);      /* task from left proc */
}
if ((buffer == full) && (!busy)) {     /* get next task */
   task = buffer;                      /* get task*/
   buffer = empty;                     /* set buffer empty */
   busy = TRUE;                        /* set process busy */
}
nrecv(Pi+1, request_tag, request);     /* check message from right */
if (request && (buffer == full)) {
   send(&buffer, Pi+1);                /* shift task forward */
   buffer = empty;
}
if (busy) {                            /* continue on current task */
   Do some work on task.
   If task finished, set busy to false.
}
```

In this code, a combined `sendrecv()` might be applied if available rather than a `send()`/`recv()` pair.

A nonblocking `nrecv()` is necessary to check for a request being received from the right.

In our pseudocode, we have simply added the parameter `request`, which is set to TRUE if a message has been received.

Nonblocking Receive Routines

PVM

Nonblocking receive, `pvm_nrecv()`, returned a value that is zero if no message has been received.

A probe routine, `pvm_probe()`, could be used to check whether a message has been received without actual reading the message

Subsequently, a normal `recv()` routine is needed to accept and unpack the message.

MPI

Nonblocking receive, `MPI_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 at that point (`MPI_Wait()` and `MPI_Test()`, respectively).

In effect, the nonblocking receive, `MPI_Irecv()`, posts a request for message and returns immediately.

Tree Structure

Extension 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.

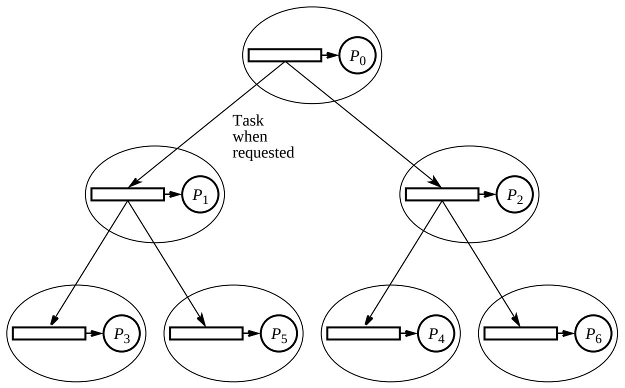


Figure 7.8 Load balancing using a tree.

Distributed Termination Detection Algorithms

Termination Conditions

In general, distributed termination at time *t* requires the following conditions to be satisfied:

- Application-specific local termination conditions exist throughout the collection of processes, at time *t*.
- There are no messages in transit between processes at time *t*.

Subtle difference between these termination conditions and those given for a centralized load-balancing system is having to take into account messages in transit.

The second condition is necessary for the distributed termination system because a message in transit might restart a terminated process.

One could imagine a process reaching its local termination condition and terminating while a message is being sent to it from another process.

Second condition is more difficult to recognize. The time that it takes for messages to travel between processes will not be known in advance.

One could conceivably wait a long enough period to allow any message in transit to arrive.

This approach would not be favored and would not permit portable code on different architectures.

Using Acknowledgment Messages

Each process is in one of two states:

1. Inactive
2. Active

Initially, without any task to perform, the process is in the inactive state. Upon receiving a task from a process, it changes to the active state.

The process that sent the task to make it enter the active state becomes its "parent."

If the process passes on a task to an inactive process, it similarly becomes the parent of this process, thus creating a tree of processes, each with a unique parent.

On every occasion when a process sends a task to another process, it expects an acknowledgment message from that process.

On every occasion when it receives a task from a process, it immediately sends an acknowledgment message, except if the process it receives the task from is its parent process.

It only sends an acknowledgment message to its parent when it is ready to become inactive. It becomes inactive when

- Its local termination condition exists (all tasks are completed).
- It has transmitted all its acknowledgments for tasks it has received.
- It has received all its acknowledgments for tasks it has sent out.

The last condition means that a process must become inactive before its parent process. When the first process becomes idle, the computation can terminate.

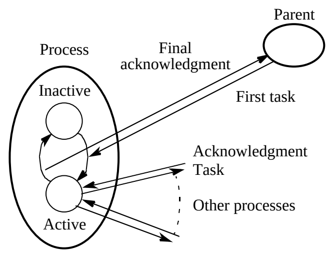


Figure 7.9 Termination using message acknowledgments.

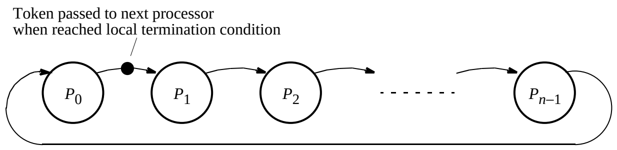
Ring Termination Algorithms

Single-pass ring termination algorithm

1. When P_0 has terminated, it generates a token that is passed to P_1.
2. When P_i ($1 \leq i < n$) receives the token and has already terminated, it passes the token onward to P_{i+1}. Otherwise, it waits for its local termination condition and then passes the token onward. P_{n-1} passes the token to P_0.
3. When P_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 a process cannot be reactivated after reaching its local termination condition.

This does not apply to work pool problems in which a process can pass a new task to an idle process



Token passed to next processor when reached local termination condition
P_0
P_1
P_2
P_{n-1}

Figure 7.10 Ring termination detection algorithm.

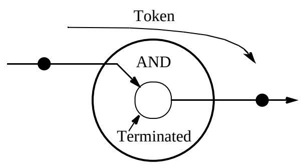


Figure 7.11 Process algorithm for local termination.

Dual-Pass Ring Termination Algorithm

Can handle processes being reactivated but requires two passes around the ring. The reason for reactivation is for process P_i, to pass a task to P_j where $j < i$ and after a token has passed P_j,. If this occurs, the token must recirculate through the ring a second time.

To differentiate these circumstances, tokens are colored white or black.

Processes are also colored white or black.

Receiving a black token means that global termination may not have occurred and the token must be recirculated around the ring again.

The algorithm is as follows, again starting at P_0:

1. P_0 becomes white when it has terminated and generates a white token to P_1.
2. The token is passed through the ring from one process to the next when each process has terminated, but the color of the token may be changed. If P_i passes a task to P_j where $j < i$ (that is, before this process in the ring), it becomes a *black process*; otherwise it is a *white process*. A black process will color a token black and pass it on. A white process will pass on the token in its original color (either black or white). After P_i has passed on a token, it becomes a white process. P_{n-1} passes the token to P_0.
3. When 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.

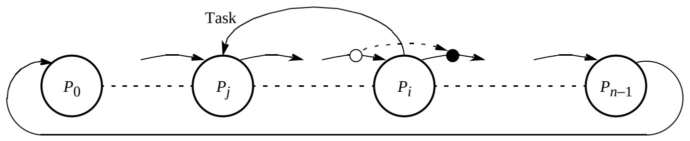


Figure 7.12 Passing task to previous processes.

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.

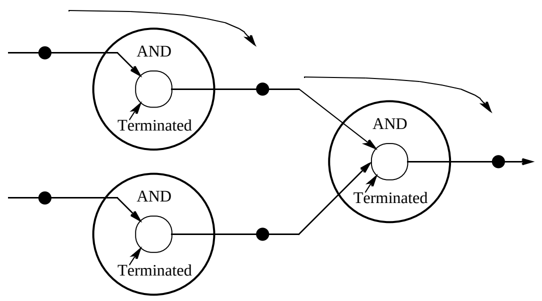


Figure 7.13 Tree termination.

Fixed Energy Distributed Termination Algorithm

Uses the notation of a fixed quantity within the system, colorfully termed "energy."

This energy is similar to a token but has a numeric value.

The system starts with all the energy being held by one process, the master process.

The master process passes out portions of the energy with the tasks to processes making requests for tasks.

Similarly, if these processes receive requests for tasks, the energy is divided further and passed to these processes.

When a process becomes idle, it passes the energy it holds back before requesting a new task.

This energy could be passed directly back to the master process or to the process giving it the original task.

A process will not hand back its energy until all the energy it handed out is returned and combined to the total energy held.

When all the energy is returned to the root and the root becomes idle, all the processes must be idle and the computation can terminate.

A significant disadvantage of the fixed energy method is that dividing the energy will be of finite precision and adding the partial energies may not equate to the original energy if floating point arithmetic is used. In addition, one can only divide the energy so far before it becomes essentially zero.

Shortest Path Problem

Finding the shortest distance between two points on a graph.

It 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)
4. The best way to climb a mountain given a terrain map with contours
5. The best route through a computer network for minimum message delay (the vertices represent computers, and the weights represent the delay between two computers)
6. The most efficient manufacturing system, where the weights represent hours of work

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

Example: The Best Way to Climb a Mountain

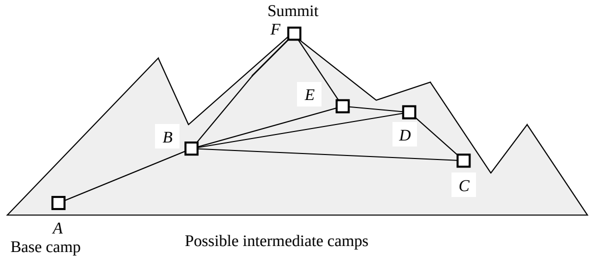


Figure 7.14 Climbing a mountain.

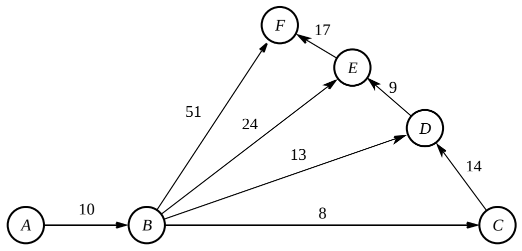


Figure 7.15 Graph of mountain climb.

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, 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 an edge and the corresponding weights associated with the edges

Adjacency matrix used for dense graphs. The adjacency list is used for sparse graphs.

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 all.
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.

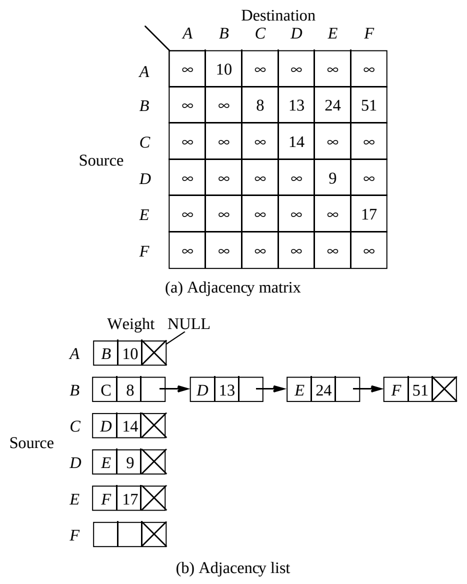


Figure 7.16 Representing a graph.

Searching a Graph

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

- Moore’s single-source shortest-path algorithm (Moore, 1957)
- Dijkstra’s single-source shortest-path algorithm (Dijkstra, 1959)

which are similar.

Moore’s algorithm is chosen because it is more amenable to parallel implementation although it may do more work.

The weights must all be positive values for the algorithm to work. (Other algorithms exist that will work with both positive and negative weights.)

Moore’s Algorithm

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

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

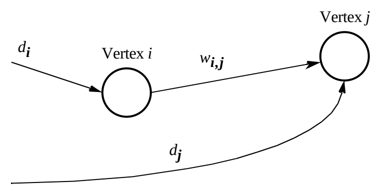


Figure 7.17 Moore’s shortest-path algorithm.

Date Structures and Code

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

Vertices are considered only when they are in the vertex queue.

Initially, only the source vertex is in the queue.

Another structure is needed to hold the current shortest distance from the source vertex to each of the other vertices.

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

The current shortest distance from the source vertex to vertex *i* will be stored in the array `dist[i]` ($1 \le i < n$).

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

Suppose `w[i][j]` holds the weight of the edge from vertex *i* and vertex *j* (infinity if no edge). The code could be of the form

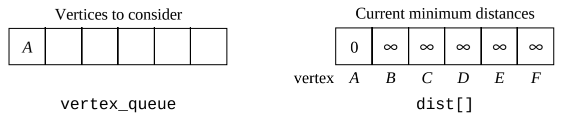
```
newdist_j = dist[i] + w[i][j];
if (newdist_j < dist[j]) dist[j] = newdist_j;
```

When a shorter distance is found to vertex *j*, vertex *j* is added to the queue (if not already in the queue), which will cause vertex *j* to be examined again. (This is an important aspect of this algorithm, which is not present in Dijkstra’s algorithm.)

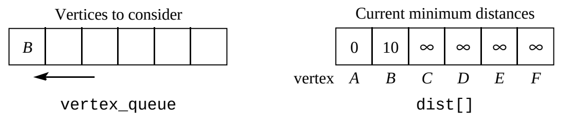
Stages in Searching a Graph

To see how this algorithm proceeds from the source vertex, let us follow the steps using our mountain climbing graph as the example.

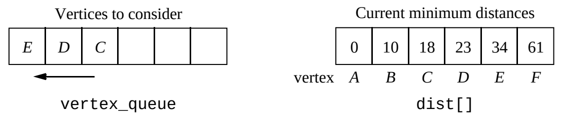
The initial values of the two key data structures are



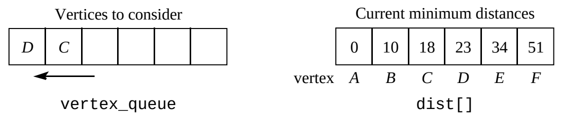
After examining A to B:



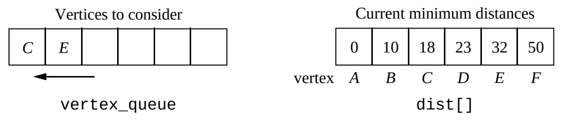
After examining B to F, E, D, and C::



After examining E to F

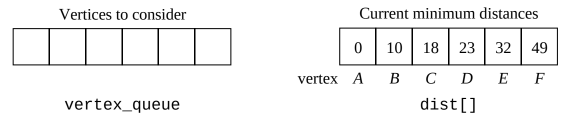


After examining D to E:



After examining C to D: No changes.

After examining E (again) to F :



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 $A \rightarrow B \rightarrow D \rightarrow E \rightarrow F$.

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[][]`, which is accessed sequentially to find the next edge.

The sequential code could then be of the form

```
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 structure holding the graph weights is fixed, this structure could be copied into each slave. We will assume a copied adjacency matrix.

Distance array, `dist[]`, is held centrally and simply copied with the vertex in its entirety.

Master

```
while (vertex_queue() != empty) {
  recv(P_ANY, source = P_i);              /* request task from slave */
  v = get_vertex_queue();
  send(&v, P_i);                          /* send next vertex and */
  send(&dist, &n, P_i);                   /* current dist array */
    .
  recv(&j, &dist[j], P_ANY, source = P_i);/* new distance */
  append_queue(j, dist[j]);               /* append vertex to queue */
                                          /* and update distance array */
};
recv(P_ANY, source = P_i);                /* request task from slave */
send(P_i, termination_tag);               /* termination message*/
```

Slave (process *i*)

```
send(P_master);                           /* send request for task */
recv(&v, P_master, tag);                  /* get vertex number */
if (tag != termination_tag) {
  recv(&dist, &n, P_master);              /* and dist array */
  for (j = 1; j < n; j++)                 /* get next edge */
    if (w[v][j] != infinity) {            /* if an edge */
      newdist_j = dist[v] + w[v][j];
      if (newdist_j < dist[j]) {
        dist[j] = newdist_j;
        send(&j, &dist[j], P_master);     /* add vertex to queue */
      }                                     /* send updated distance */
    }
}
```

Decentralized Work Pool

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 `dist[]` 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[i]` changes, process *i* will be reactivated to search again.

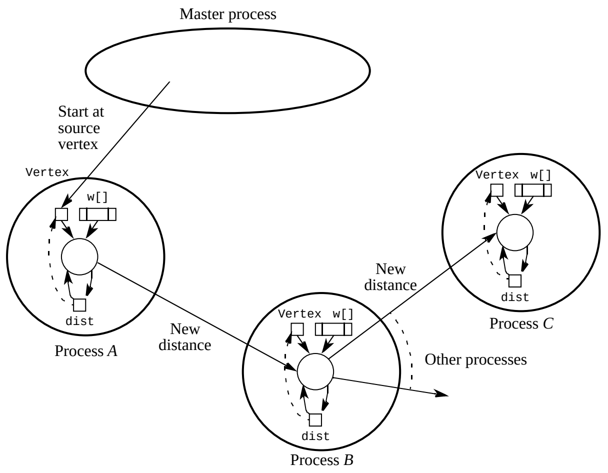


Figure 7.18 Distributed graph search.

A code segment for the slave processes might take the form

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:

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.

PROBLEMS

Scientific/Numerical

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. Develop a more efficient work pool approach that keeps processes more active.

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.

Real Life

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 of more links removed. One form of incomplete hypercube consists of two interconnected complete hypercubes of size 2^n and 2^k ($1 \le k \le n$). 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).

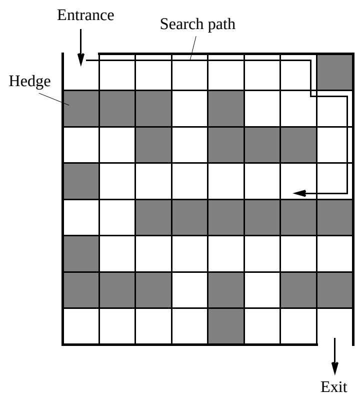


Figure 7.19 Sample maze for Problem 7-9.

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 vertices between the rooms, as illustrated in Figure 7.21. Write a program that will find the path from the outside door to the chamber holding the gold. Notice that edges are bidirectional, and cycles may exist in the graph.

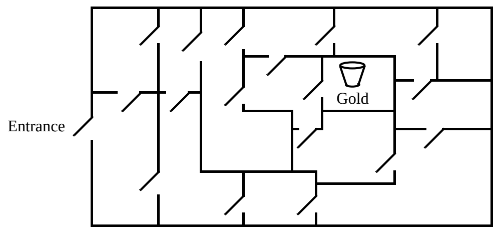


Figure 7.20 Plan of rooms for Problem 7-10.

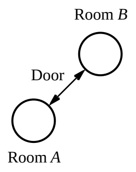


Figure 7.21 Graph representation for Problem 7-10.