

## Additional material for Chapter 6 -7

# Asynchronous Computations

Computations in which individual processes operate without needing to synchronize with other processes.

In 1st edition of textbook, asynchronous computations not dealt with except under load balancing and termination detection (Chapter 7)

Asynchronous computations important because synchronizing processes is an expensive operation which very significantly slows the computation - A major cause for reduced performance of parallel programs is due to the use of synchronization.

Global synchronization is done with barrier routines. Barriers cause processor to wait sometimes needlessly.

# Asynchronous Example

## Heat Distribution Problem Re-visited

To solve heat distribution problem, solution space divided into a two-dimensional array of points. The value of each point computed by taking average of four points around it repeatedly until values converge on the solution to a sufficient accuracy.



The waiting can be reduced by not forcing synchronization at each iteration.

## Sequential code

```
do {  
    for (i = 1; i < n; i++)  
        for (j = 1; j < n; j++)  
            g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] + h[i][j-1] + h[i][j+1]);  
  
    for (i = 1; i < n; i++)      /* find max divergence/update points */  
        for (j = 1; j < n; j++) {  
            dif = h[i][j] - g[i][j];  
            if (dif < 0) dif = -dif;  
            if (dif < max_dif) max_dif = dif;  
            h[i][j] = g[i][j];  
        }  
} while (max_dif > tolerance);    /* test convergence */
```

First section of code computing the next iteration values based on the immediate previous iteration values is traditional Jacobi iteration method.

Suppose however, processes are to continue with the next iteration before other processes have completed.

Then, the processes moving forward would use values computed from not only the previous iteration but maybe from earlier iterations.

Method then becomes an *asynchronous iterative method*.

# Asynchronous Iterative Method - Convergence

Mathematical conditions for convergence may be more strict.

Each process may not be allowed to use any previous iteration values if the method is to converge.

## Chaotic Relaxation

A form of asynchronous iterative method introduced by Chazan and Miranker (1969) in which the conditions are stated as “there must be a fixed positive integer  $s$  such that, in carrying out the evaluation of the  $i$ th iterate, a process cannot make use of any value of the components of the  $j$ th iterate if  $j < i - s$ ” (Baudet, 1978).

The final part of the code, checking for convergence of every iteration can also be reduced. It may be better to allow iterations to continue for several iterations before checking for convergence.

## Overall Parallel Code

Each process allowed to perform  $s$  iterations before being synchronized and also to update the array as it goes. At  $s$  iterations, maximum divergence recorded. Convergence is checked then.

The actual iteration corresponding to the elements of the array being used at any time may be from an earlier iteration but only up to  $s$  iterations previously. There may be a mixture of values of different iterations as the array is updated without synchronizing with other processes - truly a **chaotic** situation.



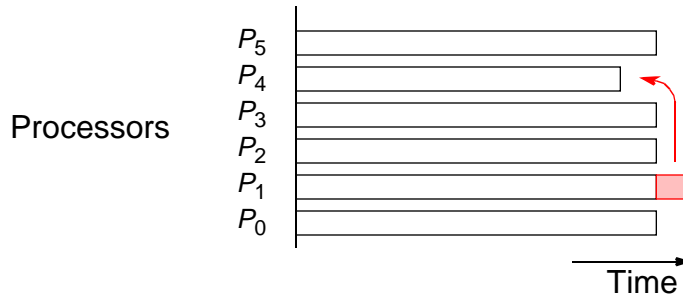
## Chapter 7

# Load Balancing and Termination Detection

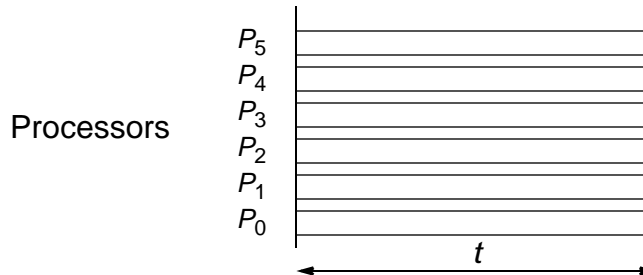
*Load balancing* – used to distribute computations fairly across processors in order to obtain the highest possible execution speed.

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

# Load balancing



(a) Imperfect load balancing leading to increased execution time



(b) Perfect 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

# Static Load Balancing

Balance load prior to the execution. Various static load-balancing algorithms.

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

Vary load 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

Computation will be divided into *work* or *tasks* to be performed, and *processes* perform these tasks. *Processes* are mapped onto *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

Can be classified as:

- Centralized
- Decentralized



## Centralized dynamic load balancing

Tasks handed out from a centralized location. Master-slave structure.

## 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).

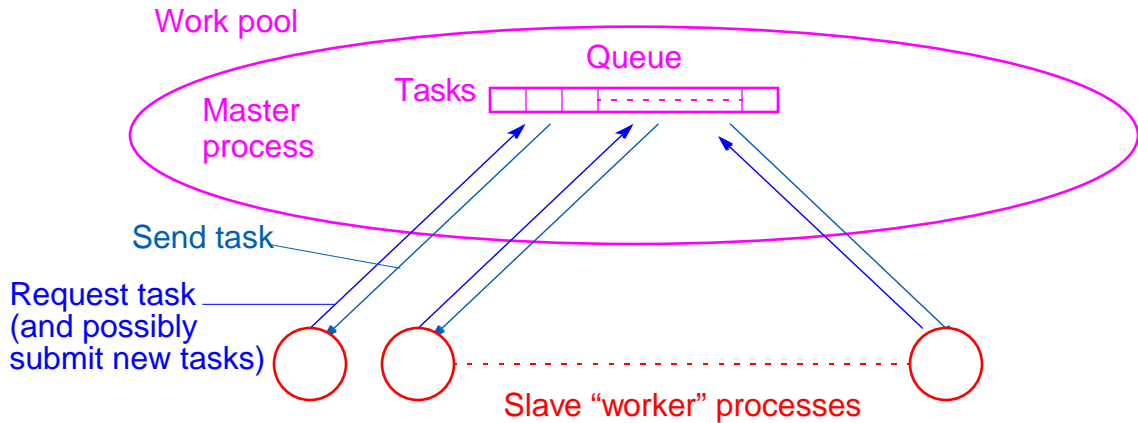
## Centralized Dynamic Load Balancing

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.*

# Centralized work pool



# Termination

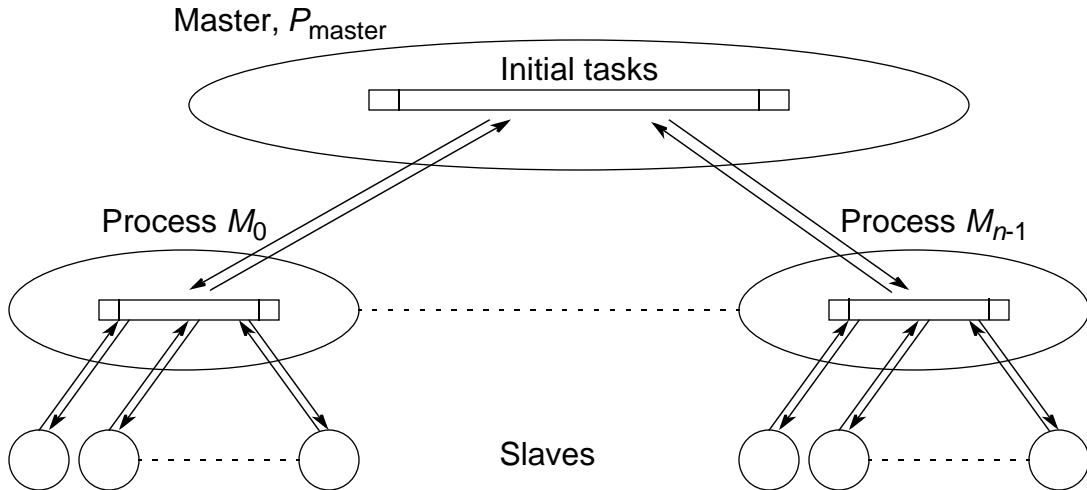
Computation terminates when:

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

*Not sufficient* to terminate when task queue empty if one or more processes are still running if a running process may provide new tasks for task queue.

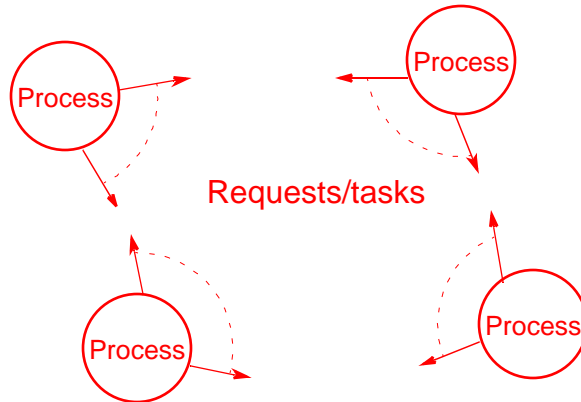
# Decentralized Dynamic Load Balancing

## Distributed Work Pool



# Fully Distributed Work Pool

Processes to execute tasks from each other



# 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.

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

Unfortunately, it can be expensive to determine process loads.

## Sender-Initiated Method

A process sends tasks to other processes it selects.

Typically, 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.

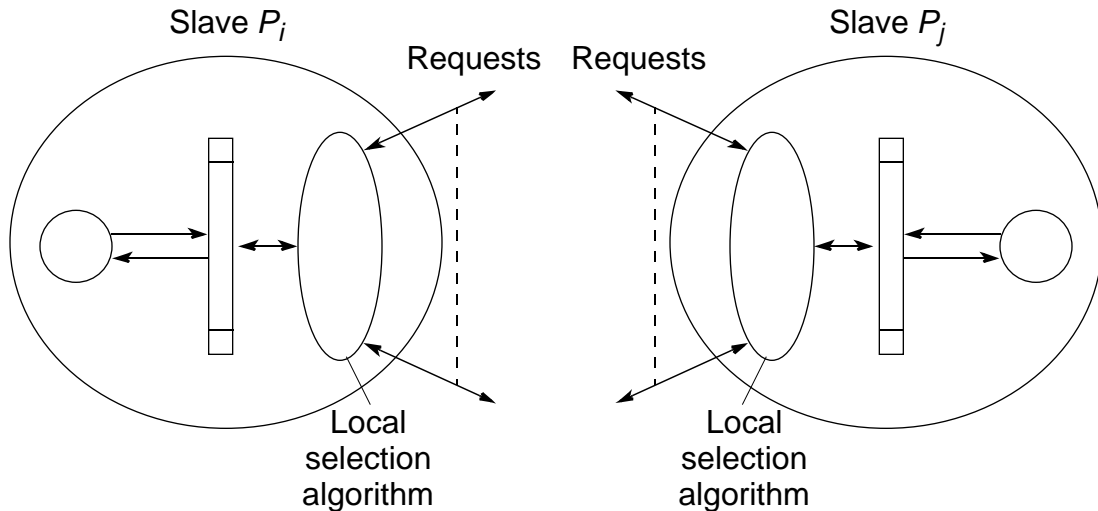
Another option is to have a mixture of both methods.

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.



# Decentralized selection algorithm requesting tasks between slaves



# Process Selection

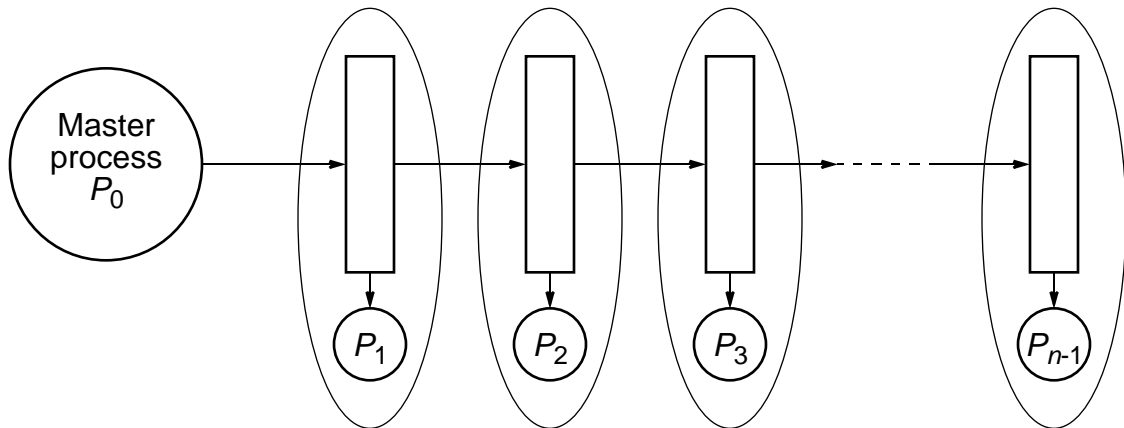
Algorithms for selecting a process:

*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 ( $n$  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  $i$ ).

# Load Balancing Using a Line 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.

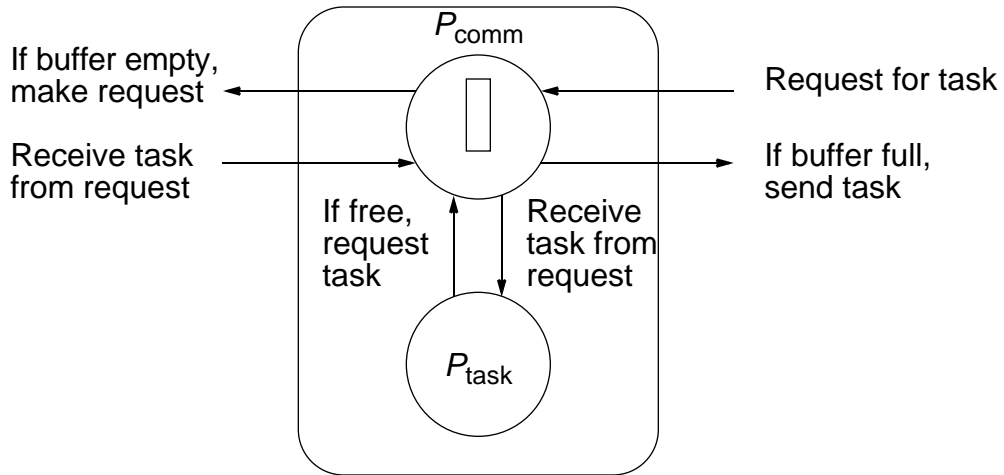
Eventually, all processes will have a task and the queue is filled with new tasks.

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

# Shifting Actions

could be orchestrated by using messages between adjacent processes:

- For left and right communication
- For the current task



## 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, P1, task_tag);      /* send tasks into queue */  
}  
recv(P1, request_tag);              /* request for task */  
send(&empty, P1, task_tag);        /* end of tasks */
```

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

```

if (buffer == empty) {
    send( $P_{i-1}$ , request_tag);      /* request new task */
    recv(&buffer,  $P_{i-1}$ , 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( $P_{i+1}$ , request_tag, request); /* check msg from right */
if (request && (buffer == full)) {
    send(&buffer,  $P_{i+1}$ );          /* shift task forward */
    buffer = empty;
}
if (busy) {                      /* continue on current task */
    Do some work on task.
    If task finished, set busy to false.
}

```

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

# 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.



# Nonblocking Receive Routines

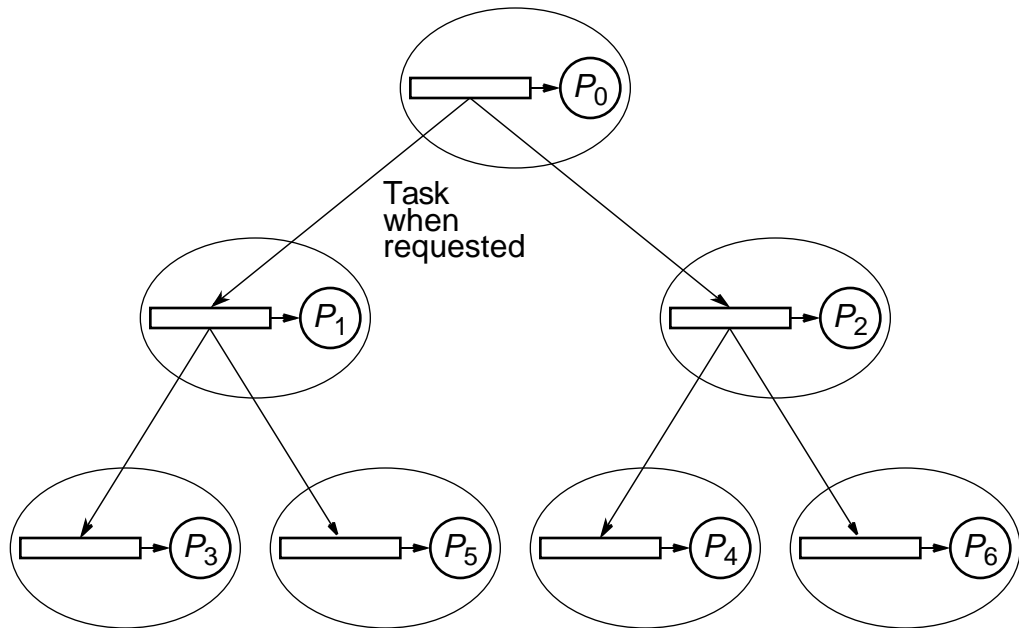
## MPI

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.

# Load balancing using a tree

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



# Distributed Termination Detection Algorithms

## Termination Conditions

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.

Second condition necessary because a message in transit might restart a terminated process. More difficult to recognize. The time that it takes for messages to travel between processes will not be known in advance.

# One very general distributed termination algorithm

Each process in one of two states:

1. Inactive - without any task to perform
2. Active

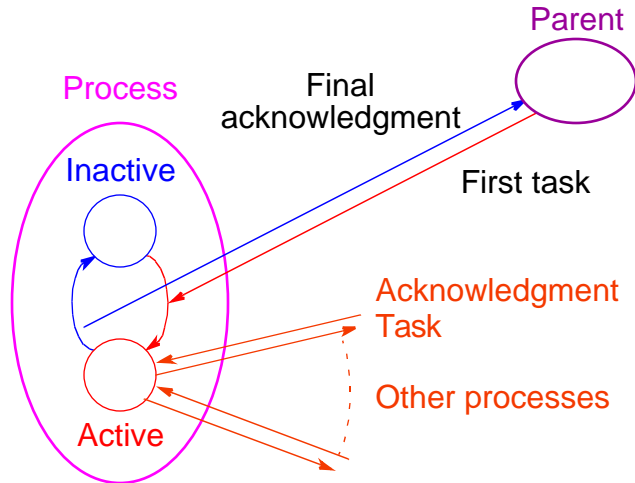
Process that sent task to make it enter the active state becomes its  
“parent.”

When process receives a task, it immediately sends an acknowledgment message, **except if the process it receives the task from is its parent process**. Only sends an acknowledgment message to its parent when it is ready to become inactive, i.e. when

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

A process must become inactive before its parent process. When first process becomes idle, the computation can terminate.

# Termination using message acknowledgments



Other termination algorithms in textbook.

# Ring Termination Algorithms

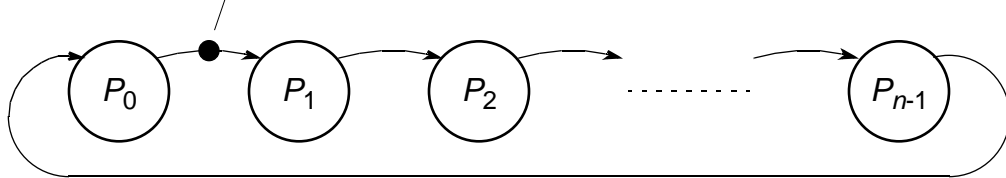
## Single-pass ring termination algorithm

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

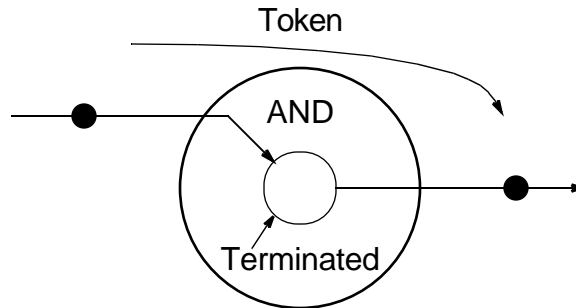
# Ring termination detection algorithm

Token passed to next processor  
when reached local termination condition





# 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 colored white or black. Processes are also colored white or black.

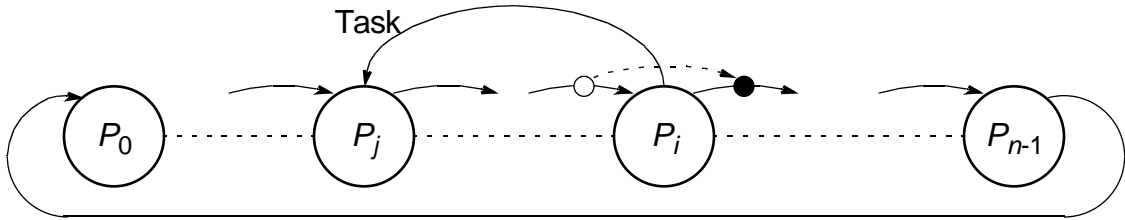
Receiving a black token means that global termination may not have occurred and token must be recirculated around ring again.

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.

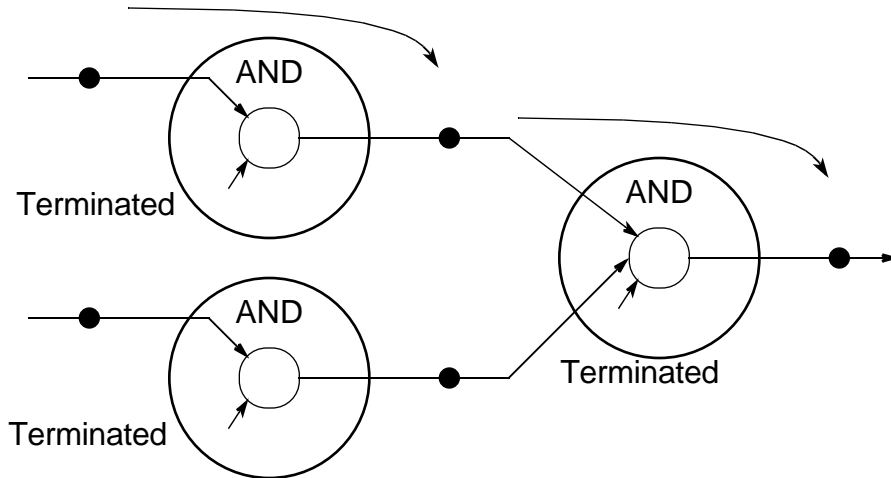
Notice that in both ring algorithms,  $P_0$  becomes the central point for global termination. Also, assumed that an acknowledge signal is generated to each request.

# Passing task to previous processes



# Tree Algorithm

Local actions described can be applied to various structures, notably a tree structure, to indicate that processes up to that point have terminated.



# Fixed Energy Distributed Termination Algorithm

A fixed quantity within system, colorfully termed “energy.”

- System starts with all the energy being held by one process, the root process.
- Root process passes out portions of energy with tasks to processes making requests for tasks.
- 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.
- 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 returned to root and the root becomes idle, all the processes must be idle and the computation can terminate.

Significant disadvantage - dividing energy will be of finite precision and adding partial energies may not equate to original energy. In addition, can only divide energy so far before it becomes essentially zero.

# Load balancing/termination detection Example

## 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*.

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*.

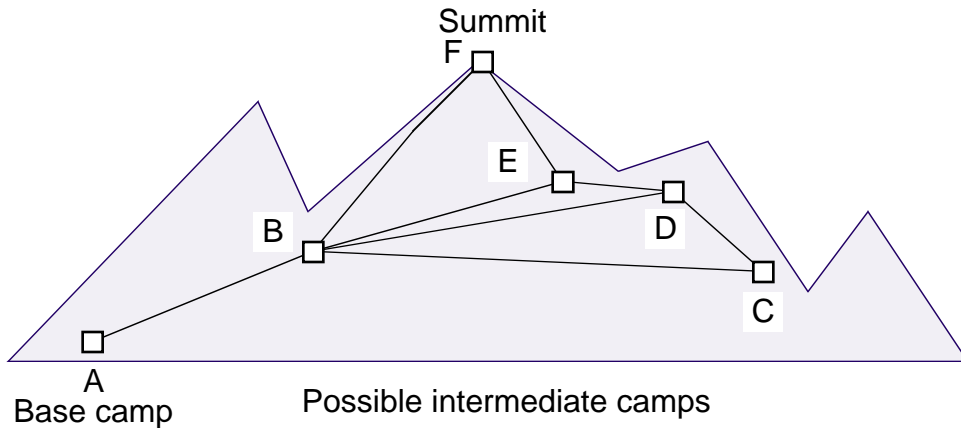
Graph could be used to find solution to many different problems; eg:

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

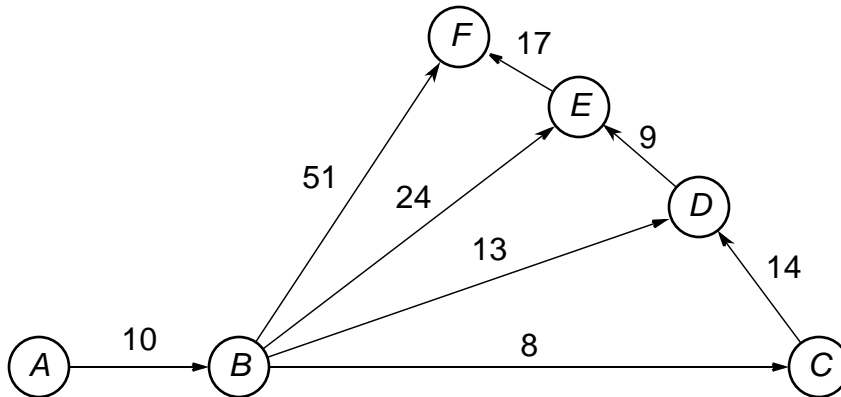
“The best way to climb a mountain” will be used as an example.



# Example: The Best Way to Climb a Mountain



# Graph of mountain climb



Weights in graph indicate 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

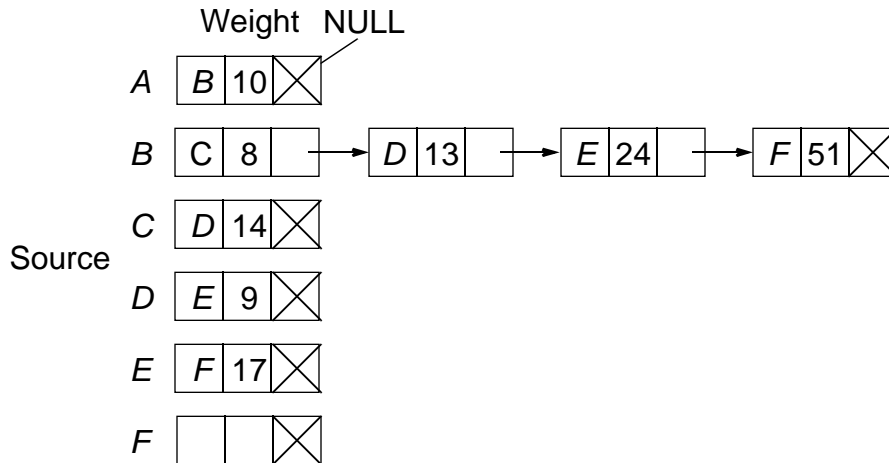
Adjacency matrix used for dense graphs. The adjacency list is used for sparse graphs.

Difference based upon space (storage) requirements. Accessing the adjacency list is slower than accessing the adjacency matrix.

# Representing the graph

		Destination					
		A	B	C	D	E	F
Source	A	•	10	•	•	•	•
	B	•	•	8	13	24	51
	C	•	•	•	14	•	•
	D	•	•	•	•	9	•
	E	•	•	•	•	•	17
	F	•	•	•	•	•	•

(a) Adjacency matrix



(b) Adjacency list

# 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.

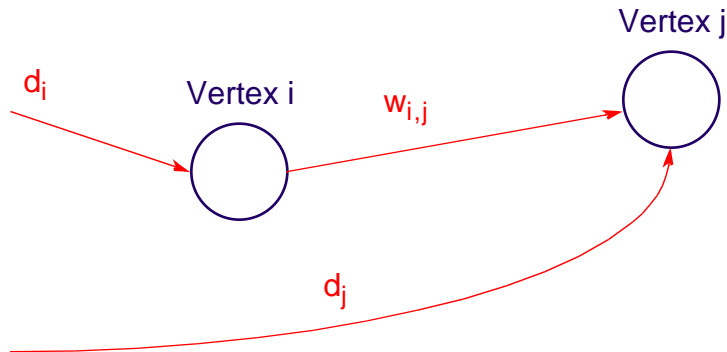
## Moore's Algorithm

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. 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  $j$ :

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

# Moore's Shortest-path Algorithm





## Date Structures

First-in-first-out vertex queue created to hold a list of vertices to examine. Initially, only source vertex is in queue.

Current shortest distance from source vertex to vertex  $i$  stored in array `dist[i]`. At first, none of these distances known and array elements are initialized to infinity.

## Code

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 - Important aspect of this algorithm, which is not present in Dijkstra's algorithm.

# Stages in Searching a Graph

## Example

The initial values of the two key data structures are

Vertices to consider

A					
---	--	--	--	--	--

vertex\_queue

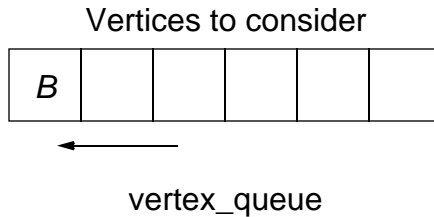
Current minimum distances

0	•	•	•	•	•
---	---	---	---	---	---

vertex   A   B   C   D   E   F

dist[]

After examining A to



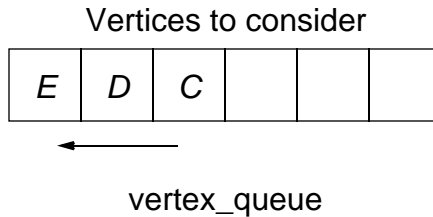
Current minimum distances

0	10	•	•	•	•
---	----	---	---	---	---

vertex   *A*   *B*   *C*   *D*   *E*   *F*

dist[]

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

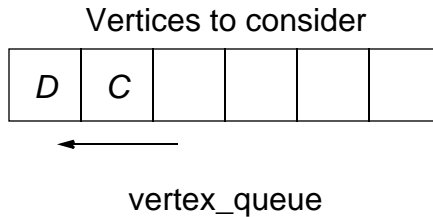


Current minimum distances

0	10	18	23	34	61
vertex $A$	$B$	$C$	$D$	$E$	$F$

dist[]

After examining  $E$  to  $F$



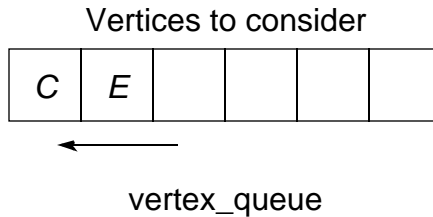
Current minimum distances

0	10	18	23	34	50
---	----	----	----	----	----

vertex    $A$     $B$     $C$     $D$     $E$     $F$

dist[]

After examining  $D$  to  $E$ :



Current minimum distances

0	10	18	23	32	50
---	----	----	----	----	----

vertex    $A$     $B$     $C$     $D$     $E$     $F$

dist[]

After examining *C* to *D*: No changes.

After examining *E* (again) to *F*:

Vertices to consider

--	--	--	--	--	--

vertex\_queue

Current minimum distances

0	10	18	23	32	49
---	----	----	----	----	----

vertex    *A*    *B*    *C*    *D*    *E*    *F*

dist[]

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 distances are recorded. The path in our case is *A*    *B*    *D*    *E*    *F*.



## Sequential Code

Let `next_vertex()` return the next vertex from the vertex queue or `no_vertex` if none.

Assume that adjacency matrix used, named `w[ ][ ]`.

```
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);                /* add to queue if not there */
      }
    }
  }                                       /*no more to consider*/
```

# Parallel Implementations

## Centralized Work Pool

Centralized work pool holds 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, say a copied adjacency matrix.

## Master

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

## Slave (process $i$ )

```

send(Pmaster);           /* send request for task */
recv(&v, Pmaster, tag);  /* get vertex number */
if (tag != termination_tag) {
    recv(&dist, &n, Pmaster); /* 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], Pmaster); /* add vertex to queue */
            } /* send updated distance */
        }
    }
}

```

## Decentralized Work Pool

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 also stores an adjacency matrix/list for vertex  $i$ , for the purpose of identifying the edges from vertex  $i$ .

# Search Algorithm

Vertex  $A$  is the first vertex to search. The process assigned to vertex  $A$  is activated.

This process will search around its vertex to find distances to connected vertices.

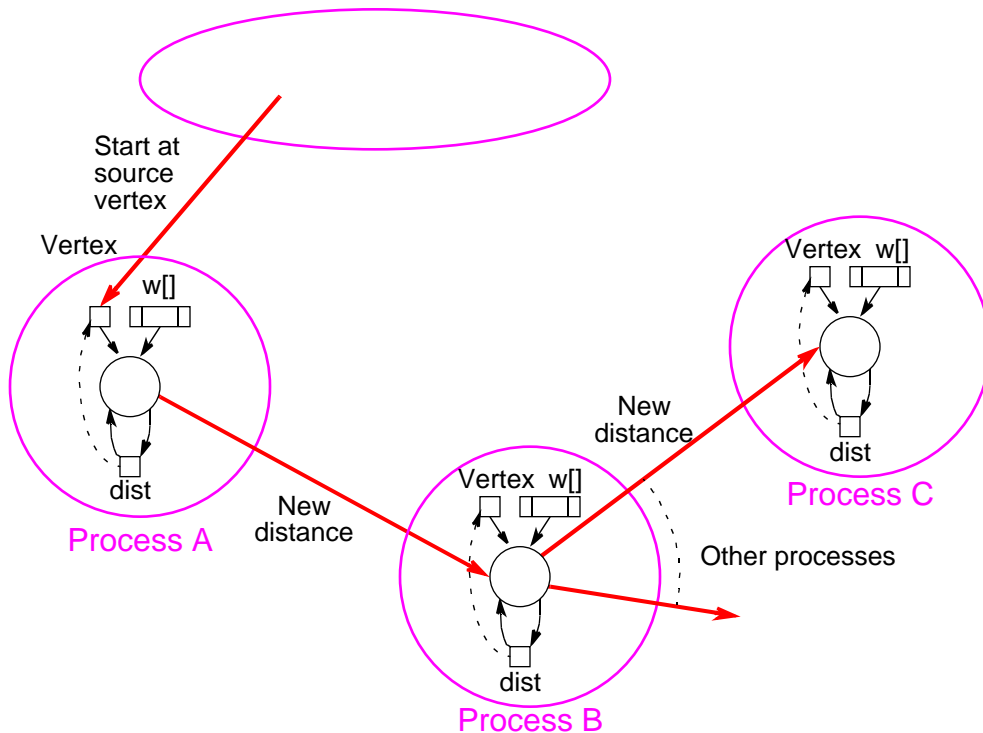
Distance to process  $j$  will be sent to process  $j$  for it to compare with its currently stored value and replace if the currently stored value is larger.

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.

# Distributed graph search

Master process



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 */
        }
```



## Simplified 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 */  
        }  
}
```

Mechanism necessary to repeat actions and terminate when all processes idle - must cope with messages in transit.

## **Simplest solution**

Use synchronous message passing, in which a process cannot proceed until the destination has received the message.

Process only active after its vertex is placed on queue. Possible for many processes to be inactive, leading to an inefficient solution.

Method also impractical for a large graph if one vertex is allocated to each processor. Group of vertices could be allocated to each processor.