Distributed Termination Detection Algorithms

Termination Conditions

In general, distributed termination at time *t* requires the following conditions to be satisfied (Bertsekas & Tsitsiklis, 1989):

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

What is the difference between these and the centralized one?

How could we detect the occurrence of those two conditions?

Using Acknowledgment Messages

(Bertsekas & Tsitsiklis, 1989) describe a distributed termination method using request and acknowledgment messages.

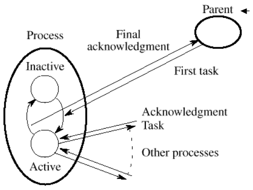
+ very general
+ mathematically sound
+ copes with messages being in transit.

Each process is in one of two states:

1. Inactive
2. Active

A task only sends an ack message to its parent when it is ready to become inactive, i.e.,:

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



The task that sent the task to make the process enter the active state.

Figure 7.9 Termination using message acknowledgments.

Ring Termination Algorithms

For termination purposes, the processes are organized in a ring structure:

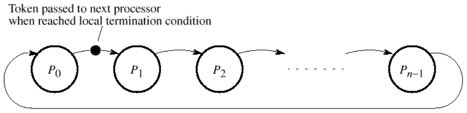


Figure 7.10 Ring termination detection algorithm.

The single-pass ring termination algorithm:

- When P_0 is terminated, it generates a token that it passes to P_1.
- When P_i receives the token and has already terminated, it passes the token onward to P_{i+1}. Otherwise it waits for local termination condition and then passes the token onward. P_{n-1} passes the token to P_0.
- When P0 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.

Each process, except the first one, implements the following function:

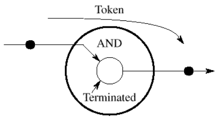


Figure 7.11 Process algorithm for local termination.

The algorithm assumes that a process cannot be reactivated after reaching its local termination condition.

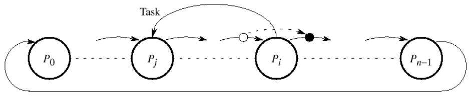


Figure 7.12 Passing task to previous processes.

•In both algorithms, P0 becomes a central point for global termination.
•An ack signal is generated to each request.

The dual-pass ring termination algorithm (Dujkstra, Feijen and Gastren, 1983) :

- can handle processes being reactivated but requires two passes around the ring. *Reason for reactivation?*
- uses two tokens: white and black.
  - Black token: global termination may not have occurred and the token must be recirculated around the ring again.

The Algorithm:

- When P_0 becomes white when it has terminated and it generates a white token that it passes to P_1.
- When P_i receives the token and has already terminated, it passes the token onward to P_{i+1}. But the color of the token may be changed (Pi to Pj where j<i then black, otherwise white)
  - A black process will color the token black and pass it on.
  - A white process will pass on the token in its original color. After Pi has passed on a token, it becomes a white process.

  P_{n-1} passes the token to P_0.
- When P_0 receives a black token, it passes on a white token; if it receives a white token, all processes have terminated.

Tree Algorithm

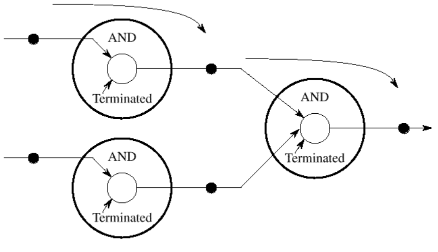


Figure 7.13 Tree termination.

Fixed Energy Distributed Termination Algorithm

- Uses the notation of a fixed quantity within the system, “energy”
  - similar to a token but has a numeric value.
- Master process passes out portions of the energy with the tasks to processes making requests for tasks.
- Similarly, it 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.
  - can pass it to the master
  - can pass it back to original task

    This creates a tree-like structure

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

One disadv: finite precision operations!

Program Example

Load balancing strategies can be used in image processing, ray tracing, colume rendering, optimization and search areas.

Shortest Path Problem

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.

- Interconnected nodes can be described by a *graph*.
- Nodes - vertices
- Links - edges
- Directed graph - if edges can only be traversed in one direction.

Graphs can be used to find the solution to many different problems.

The Best Way to Climb a Mountain

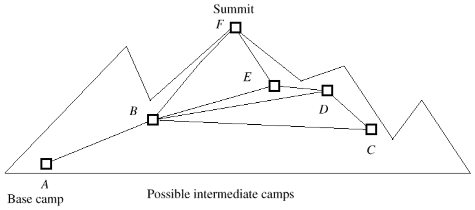


Figure 7.14 Climbing a mountain.

Graph Representation

Graphs can be represented in a program in two ways:
- Adjacency matrix
- Adjacency list

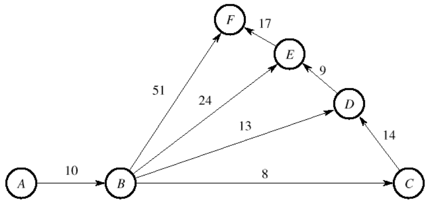
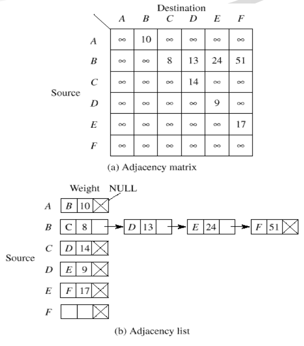


Figure 7.15 Graph of mountain climb.

One is chosen acc.to:
- graph structure
- storage requirements
- speed
- partitioning of tasks and its effect on accessing the information.



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

(a) Adjacency matrix

(b) Adjacency list

Figure 7.16 Representing a graph.

Searching A Graph

Single-source shortest-path graph algorithms find the minimum accumulation of weights from a source vertex to a destination vertex:

- Moore's algorithm (1957)
  - Although it may do more work, it is more amenable to parallel implementation (Adamson and Tick, 1992)
  - Weights must be +.
- Dijkstra's algorithm (1959)

Moore's Algorithm

Starting with the source vertex, find the distance to vertex *j* through vertex *i* and compare with the current mimimum distance to vertex *j*.

- Change the minimum distance if this path is shorter

Iterative solution: $d_j = \min(d_j, d_i + w_{ij})$ where

d_i is the current minimum distance from the source vertex to vertex *i*.

w_{ij} is the weight of the edge from vertex *i* to vertex *j*.

We can implement this formula using directed search.

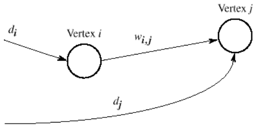


Figure 7.17 Moore's shortest-path algorithm.

Sequential Code

```
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 continue if not there*/
         }
   }                          /* no more vertices to consider*/
```

Parallel Implementation - Centralized Work Pool

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 an edge*/
      if (w[v][j] != inifinity) {
         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 */
      }
}
```

Parallel Implementation - Decentralized Work Pool

Slave (process i):

```
recv(newdist, P_ANY);
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 */
     }
```

Slave (process i):

```
recv(newdist, P_ANY);
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 */
     }
```

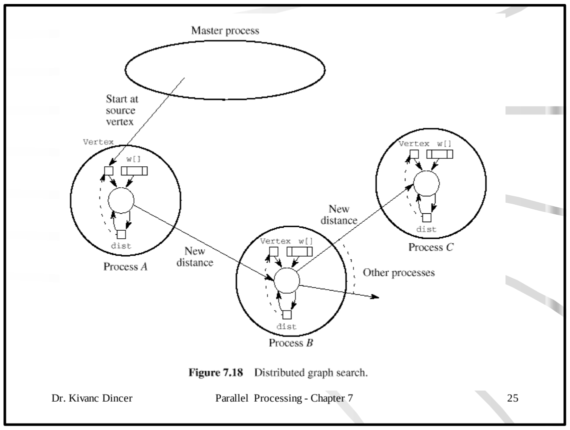


Figure 7.18 Distributed graph search.

Dr. Kivanc Dincer Parallel Processing - Chapter 7 25