Chapter 9

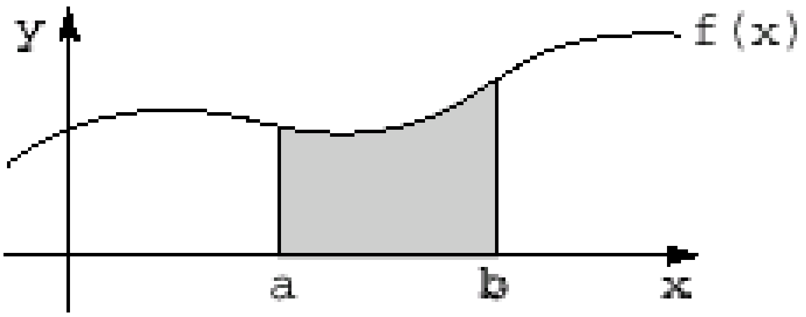
Paradigms for Process Interaction

9.1 Manager/Workers (distributed bag of tasks)

- A manager process to implement the bag, hand out tasks, collect results, and detect termination.
- Example: an adaptive quadrature using manager/worker paradigm

(from Section 1.5)

- Quadrature



Recursive parallelism (from Section 1.5)

- The basic quadrature algorithm

```
double fleft = f(a), fright, area = 0.0;
double width = (b-a) / INTERVALS;
for [x = (a + width) to b by width] {
  fright = f(x);
  area = area + (fleft + fright) * width / 2;
  fleft = fright;
}
```

Iterative Quadrature Program

(Recursive parallelism (from Section 1.5, coninued.)

- The adaptive quadrature algorithm

```
double quad(double left,right,fleft,fright,lrarea) {
  double mid = (left + right) / 2;
  double fmid = f(mid);
  double larea = (fleft+fmid) * (mid-left) / 2;
  double rarea = (fmid+fright) * (right-mid) / 2;
  if (abs((larea+rarea) - lrarea) > EPSILON) {
    # recurse to integrate both halves
    larea = quad(left, mid, fleft, fmid, larea);
    rarea = quad(mid, right, fmid, fright, rarea);
  }
  return (larea + rarea);
}
```

Recursive Procedure for Quadrature Problem

(Recursive parallelism (from Section 1.5, coninued.)

```
double quad(double left,right,fleft,fright,lrarea) {
  double mid = (left + right) / 2;
  double fmid = f(mid);
  double larea = (fleft+fmid) * (mid-left) / 2;
  double rarea = (fmid+fright) * (right-mid) / 2;
  if (abs((larea+rarea) - lrarea) > EPSILON) {
    # recurse to integrate both halves in parallel
    co larea = quad(left, mid, fleft, fmid, larea);
    // rarea = quad(mid, right, fmid, fright, rarea);
    oc
  }
  return (larea + rarea);
}
```

Recursive Parallel Adaptive Quadrature

```
module Manager
  op getTask(result double left, right);
  op putResult(double area);
body Manager
  process manager {
    double a, b;          # interval to integrate
    int numIntervals;     # number of intervals to use
    double width = (b-a)/numIntervals;
    double x = a, totalArea = 0.0;
    int tasksDone = 0;
    while (tasksDone < numIntervals) {
      in getTask(left, right) st x < b ->
           left = x; x += width; right = x;
      [] putResult(area) ->
           totalArea += area;
           tasksDone++;
      ni
    }
    print the result totalArea;
  }
end Manager
```

```
double f() { ... }          # function to integrate
double quad(...) { ... }  # adaptive quad function

process worker[w = 1 to numWorkers] {
  double left, right, area = 0.0;
  double fleft, fright, lrarea;
  while (true) {
    call getTask(left, right);
    fleft = f(left); fright = f(right);
    lrarea = (fleft + fright) * (right - left) / 2;
    # calculate area recursively as shown in Section 1.5
    area = quad(left, right, fleft, fright, lrarea);
    send putResult(area);
  }
}
```

Figure 9.2 Adaptive quadrature using manager/workers paradigm.

9.2 Heartbeat Algorithms

- The heartbeat algorithm is useful for data iterative applications.
- When the data are divided among workers, each is responsible for updating a part, and new data values depend on values held by workers or their immediate neighbors.
- Heartbeat: expand – sending information out; contract – gathering new information.

```
process Worker[ i = 1 to numWorkers ]  {
    declarations of local variables;
    initialize local variables;
    While (not done) {
        send values to neighbors;
        receive values from neighbors;
        update local values;
}  }
```

9.2 Heartbeat Algorithms

- Cellular Automata: the game of life
  - The board is initialized
  - Every cell examines its state and the state of its neighbors, then makes a state transition according to the rules:
    - A live cell with zero or one live neighbors dies from loneliness;
    - A live cell with two or three live neighbors survives for another generation;
    - A live cell with four or more live neighbors dies due to overpopulation;
    - A dead cell with exactly three live neighbors become alive.

```
chan exchange[1:n,1:n](int row, column, state);

process cell[i = 1 to n, j = 1 to n] {
  int state;   # initialize to dead or alive
  declarations of other variables;
  for [k = 1 to numGenerations] {
    # exchange state with 8 neighbors
    for [p = i-1 to i+1, q = j-1 to j+1]
      if (p != q)
        send exchange[p,q](i, j, state);
    for [p = 1 to 8] {
      receive exchange[i,j](row, column, value);
      save value of neighbor's state;
    }
    update local state using rules in text;
  }
}
```

9.3 Pipeline Algorithms

- A pipeline is a linear collection of filter processes.
- A filter process receives data from an input port, processes the data, and sends results to an output port.

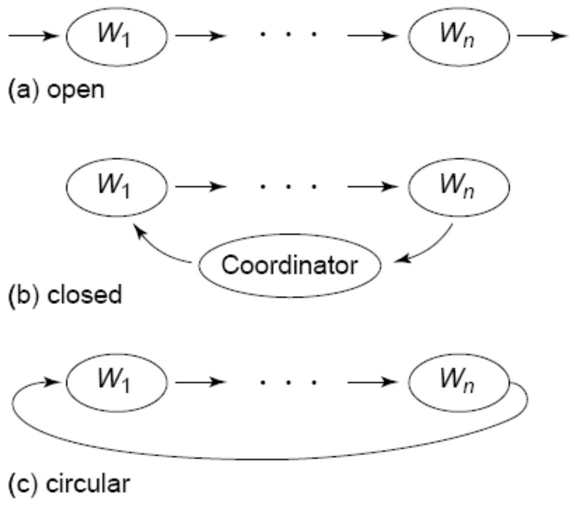


Figure 9.5 Pipeline structures for parallel computing.

9.3.1 A Distributed Matrix Multiplication Pipeline

```
chan vector[n](double v[n]);  # messages to workers
chan result(double v[n]);     # rows of c to coordinator

process Coordinator {
  double a[n,n], b[n,n], c[n,n];
  initialize a and b;
  for [i = 0 to n-1]          # send all rows of a
    send vector[0](a[i,*]);
  for [i = 0 to n-1]          # send all columns of b
    send vector[0](b[*,i]);
  for [i = n-1 to 0]          # receive rows of c
    receive result(c[i,*]);   # in reverse order
}
```

Figure 9.6 (a) Matrix multiplication pipeline: Coordinator process.

```
process Worker[w = 0 to n-1] {
  double a[n], b[n], c[n];  # my row or column of each
  double temp[n];           # used to pass vectors on
  double total;       # used to compute inner product

  # receive rows of a; keep first and pass others on
  receive vector[w](a);
  for [i = w+1 to n-1] {
    receive vector[w](temp); send vector[w+1](temp);
  }

  # get columns and compute inner products
  for [j = 0 to n-1] {
    receive vector[w](b);  # get a column of b
    if (w < n-1)    # if not last worker, pass it on
      send vector[w+1](b);
    total = 0.0;
    for [k = 0 to n-1]     # compute one inner product
      total += a[k] * b[k];
    c[j] = total;          # put total into c
  }
```

```
  # send my row of c to next worker or coordinator
  if (w < n-1)
    send vector[w+1](c);
  else
    send result(c);
  # receive and pass on earlier rows of c
  for [i = 0 to w-1] {
    receive vector[w](temp);
    if (w < n-1)
      send vector[w+1](temp);
    else
      send result(temp);
  }
}
```

Worker processes

Matrix multiplication by blocks

Matrix a:

a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}

Matrix b:

b_{11}	b_{12}	b_{13}	b_{14}
b_{21}	b_{22}	b_{23}	b_{24}
b_{31}	b_{32}	b_{33}	b_{34}
b_{41}	b_{42}	b_{43}	b_{44}

$c_{ij} = a_{i1}*b_{1j} + a_{i2}*b_{2j} + a_{i3}*b_{3j} + a_{i4}*b_{4j}$

Matrix multiplication by blocks

a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}

b_{11}	b_{12}	b_{13}	b_{14}
b_{21}	b_{22}	b_{23}	b_{24}
b_{31}	b_{32}	b_{33}	b_{34}
b_{41}	b_{42}	b_{43}	b_{44}

$c_{11} = a_{11}*b_{11} + a_{12}*b_{21} + a_{13}*b_{31} + a_{14}*b_{41}$

For **worker$_{11}$** who computes c_{11}, it needs to get every element in row 1 and every element in column 1. Initially, worker$_{11}$ holds a[1,1] and b[1,1]. If we shift row 1 of a to left one column and shift column 1 of b up one row, worker$_{11}$ will get a[1,2] and b[2,1]. Doing so repeatedly 4 times, worker$_{11}$ will get all it needs.
However, this method works for the diagonal workers **only**.

Matrix multiplication by blocks

a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}

$\mathbf{a_{12}}$	$\mathbf{a_{13}}$	$\mathbf{a_{14}}$	$\mathbf{a_{11}}$
$\mathbf{a_{23}}$	$\mathbf{a_{24}}$	$\mathbf{a_{21}}$	$\mathbf{a_{22}}$
$\mathbf{a_{34}}$	$\mathbf{a_{31}}$	$\mathbf{a_{32}}$	$\mathbf{a_{33}}$
$\mathbf{a_{41}}$	$\mathbf{a_{42}}$	$\mathbf{a_{43}}$	$\mathbf{a_{44}}$

Rearrange **a** by shifting row i of a circularly left *i* columns.

Matrix multiplication by blocks

b_{11}	b_{12}	b_{13}	b_{14}
b_{21}	b_{22}	b_{23}	b_{24}
b_{31}	b_{32}	b_{33}	b_{34}
b_{41}	b_{42}	b_{43}	b_{44}

$\mathbf{b_{21}}$	$\mathbf{b_{32}}$	$\mathbf{b_{43}}$	$\mathbf{b_{14}}$
$\mathbf{b_{31}}$	$\mathbf{b_{42}}$	$\mathbf{b_{13}}$	$\mathbf{b_{24}}$
$\mathbf{b_{41}}$	$\mathbf{b_{12}}$	$\mathbf{b_{23}}$	$\mathbf{b_{34}}$
$\mathbf{b_{11}}$	$\mathbf{b_{22}}$	$\mathbf{b_{33}}$	$\mathbf{b_{44}}$

Rearrange **b** by shifting column j circularly up *j* rows.

Matrix multiplication by blocks

a_{12}	a_{13}	a_{14}	a_{11}
a_{23}	a_{24}	a_{21}	a_{22}
a_{34}	a_{31}	a_{32}	a_{33}
a_{41}	a_{42}	a_{43}	a_{44}

b_{21}	b_{32}	b_{43}	b_{14}
b_{31}	b_{42}	b_{13}	b_{24}
b_{41}	b_{12}	b_{23}	b_{34}
b_{11}	b_{22}	b_{33}	b_{44}

Putting rearranged **a** and **b** together (see Page 442).

a_{12}, b_{21}	a_{13}, b_{32}	a_{14}, b_{43}	a_{11}, b_{14}
a_{23}, b_{31}	a_{24}, b_{42}	a_{21}, b_{13}	a_{22}, b_{24}
a_{34}, b_{41}	a_{31}, b_{12}	a_{32}, b_{23}	a_{33}, b_{34}
a_{41}, b_{11}	a_{42}, b_{22}	a_{43}, b_{33}	a_{44}, b_{44}

Each worker computes the product of the two holding values

$c_{12} = a_{11}*b_{12} + a_{12}*b_{22} + \mathbf{a_{13}*b_{32}} + a_{14}*b_{42}$

Matrix multiplication by blocks

a_{13}	a_{14}	a_{11}	a_{12}
a_{24}	a_{21}	a_{22}	a_{23}
a_{31}	a_{32}	a_{33}	a_{34}
a_{42}	a_{43}	a_{44}	a_{41}

b_{31}	b_{42}	b_{13}	b_{24}
b_{41}	b_{12}	b_{23}	b_{34}
b_{11}	b_{22}	b_{33}	b_{44}
b_{21}	b_{32}	b_{43}	b_{14}

Shift **a** toward left one column, and shift **b** up one row.

a_{13}, b_{31}	a_{14}, b_{42}	a_{11}, b_{13}	a_{12}, b_{24}
a_{24}, b_{41}	a_{21}, b_{12}	a_{22}, b_{23}	a_{23}, b_{34}
a_{31}, b_{11}	a_{32}, b_{22}	a_{33}, b_{33}	a_{34}, b_{44}
a_{42}, b_{21}	a_{43}, b_{32}	a_{44}, b_{43}	a_{41}, b_{14}

Each worker compute a product of the two holding values.

$c_{12} = a_{11}*b_{12} + a_{12}*b_{22} + a_{13}*b_{32} + \mathbf{a_{14}*b_{42}}$

Matrix multiplication by blocks

a_{14}	a_{11}	a_{12}	a_{13}
a_{21}	a_{22}	a_{23}	a_{24}
a_{32}	a_{33}	a_{34}	a_{31}
a_{43}	a_{44}	a_{41}	a_{42}

b_{41}	b_{12}	b_{23}	b_{34}
b_{11}	b_{22}	b_{33}	b_{44}
b_{21}	b_{32}	b_{43}	b_{14}
b_{31}	b_{42}	b_{13}	b_{24}

Shift **a** toward left one column, and shift **b** up one row.

a_{14} , b_{41}	a_{11} , b_{12}	a_{12} , b_{23}	a_{13} , b_{34}
a_{21} , b_{11}	a_{22} , b_{22}	a_{23} , b_{33}	a_{24} , b_{44}
a_{32} , b_{21}	a_{33} , b_{32}	a_{34} , b_{43}	a_{31} , b_{14}
a_{43} , b_{31}	a_{44} , b_{42}	a_{41} , b_{13}	a_{42} , b_{24}

Each worker compute a product of the two holding values.

$c_{12} = \mathbf{a_{11}*b_{12}} + a_{12}*b_{22} + a_{13}*b_{32} + a_{14}*b_{42}$

Matrix multiplication by blocks

a_{11}	a_{12}	a_{13}	a_{14}
a_{22}	a_{23}	a_{24}	a_{21}
a_{33}	a_{34}	a_{31}	a_{32}
a_{44}	a_{41}	a_{42}	a_{43}

b_{11}	b_{22}	b_{33}	b_{44}
b_{21}	b_{32}	b_{43}	b_{14}
b_{31}	b_{42}	b_{13}	b_{24}
b_{41}	b_{12}	b_{23}	b_{34}

Shift **a** toward left one column, and shift **b** up one row.

a_{11} , b_{11}	a_{12} , b_{22}	a_{13} , b_{33}	a_{14} , b_{44}
a_{22} , b_{21}	a_{23} , b_{32}	a_{24} , b_{43}	a_{21} , b_{14}
a_{33} , b_{31}	a_{34} , b_{42}	a_{31} , b_{13}	a_{32} , b_{24}
a_{44} , b_{41}	a_{41} , b_{12}	a_{42} , b_{23}	a_{43} , b_{34}

Each worker compute a product of the two holding values.

$c_{12} = a_{11}*b_{12} + \mathbf{a_{12}*b_{22}} + a_{13}*b_{32} + a_{14}*b_{42}$

```
chan left[1:n,1:n](double);  # for circulating a left
chan up[1:n,1:n](double);    # for circulating b up

process Worker[i = 1 to n, j = 1 to n] {
  double aij, bij, cij;
  int LEFT1, UP1, LEFTI, UPJ;
  initialize above values;

  # shift values in aij circularly left i columns
  send left[i,LEFTI](aij); receive left[i,j](aij);
  # shift values in bij circularly up j rows
  send up[UPJ,j](bij); receive up[i,j](bij);
  cij = aij * bij;

  for [k = 1 to n-1] {
    # shift aij left 1, bij up 1, then multiply and add
    send left[i,LEFT1](aij); receive left[i,j](aij);
    send up[UP1,j](bij); receive up[i,j](bij);
    cij = cij + aij*bij;
  }
}
```
Note: Elements should be replaced by blocks

Figure 9.7

9.4 Probe/Echo Algorithms

- A *probe* is a message sent by one node to its successors; and *echo* is a subsequent reply.
- The probe/echo paradigm is thus the concurrent programming analog of DFS (depth-first search).

9.4.1 Broadcast in a Network

- An efficient way to broadcast a message is to do so following a *spanning tree* of the network.

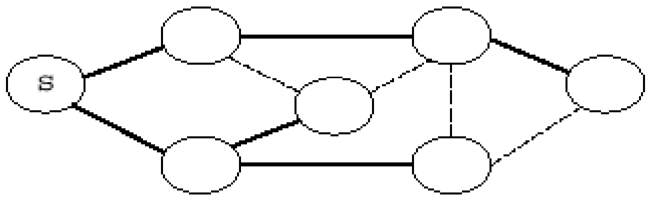


Figure 9.8 A spanning tree of a network of nodes.

(continued)

```
type graph = bool [n,n];
chan probe[n](graph spanningTree; message m);

process Node[p = 0 to n-1] {
  graph t; message m;
  receive probe[p](t, m);
  for [q = 0 to n-1 st q is a child of p in t]
    send probe[q](t, m);
}

process Initiator {  # executed on source node S
  graph topology = network topology;
  graph t = spanning tree of topology;
  message m = message to broadcast;
  send probe[S](t, m);
}
```

Figure 9.9 Network broadcast using a spanning tree.

```
chan probe[n](message m);

process Node[p = 1 to n] {
  bool links[n] = neighbors of node p;
  int num = number of neighbors;
  message m;
  receive probe[p](m);
  # send m to all neighbors
  for [q = 0 to n-1 st links[q]]
    send probe[q](m);
  # receive num-1 redundant copies of m
  for [q = 1 to num-1]
    receive probe[p](m);
}

process Initiator {  # executed on source node S
  message m = message to broadcast;
  send probe[S](m);
}
```

Figure 9.10 Broadcast using neighbor sets.

9.4.2 Computing the topology of a network

- Two phases:
  - Each node sends a probe to its neighbors
  - Each node sends an echo containing its local topology information back to the prober.
- The algorithm for tree
- The algorithm for graph

```
type graph = bool [n,n];
chan probe[n](int sender);
chan echo[n](graph topology)    # parts of the topology
chan finalecho(graph topology)  # final topology


process Initiator {
  graph topology;
  send probe[S](S)   # start probe at local node
  receive finalecho(topology);
}
```

Fig 9.11

```
process Node[p = 0 to n-1] {
  bool links[n] = neighbors of node p;
  graph newtop, localtop = ([n*n] false);
  int parent;   # node from whom probe is received
  localtop[p,0:n-1] = links;   # initially my links

  receive probe[p](parent);
  # send probe to other neighbors, who are p's children
  for [q = 0 to n-1 st (links[q] and q != parent)]
    send probe[q](p);

  # receive echoes and union them into localtop
  for [q = 0 to n-1 st (links[q] and q != parent)] {
    receive echo[p](newtop);
    localtop = localtop or newtop;  # logical or
  }
  if (p == S)
    send finalecho(localtop);    # node S is root
  else
    send echo[parent](localtop);
}
```

Fig 9.11

```
type graph = bool [n,n];
type kind = (PROBE, ECHO);
chan probe_echo[n](kind k; int sender; graph topology);
chan finalecho(graph topology);


 process Initiator {
   graph topology;   # network topology
   send probe_echo[source](PROBE, source, ∅);
   receive finalecho(topology);
 }
```

Figure 9.12 Probe/echo algorithm for computing the topology of a graph.

```
process Node[p = 0 to n-1] {
  bool links[n] = neighbors of node p;
  graph newtop, localtop = ([n*n] false);
  int first, sender; kind k;
  int need_echo = number of neighbors - 1;
  localtop[p,0:n-1] = links;   # initially my links
  receive probe_echo[p](k, first, newtop);  # get probe
  # send probe on to to all other neighbors
  for [q = 0 to n-1 st (links[q] and q != first)]
    send probe_echo[q](PROBE, p, ∅);
  while (need_echo > 0) {
    # receive echoes or redundant probes from neighbors
    receive probe_echo[p](k, sender, newtop);
    if (k == PROBE)
      send probe_echo[sender](ECHO, p, ∅);
    else  # k == ECHO {
      localtop = localtop or newtop;  # logical or
      need_echo = need_echo-1;
    }
  }
  if (p == S)
    send finalecho(localtop);
  else
    send probe_echo[first](ECHO, p, localtop);
}
```

9.5 Broadcast Algorithms

broadcast ch(m)

- If process P[i] executes **broadcast** ch(*m*)**,** one copy of *m* will be placed on every channel, ch[i], including that of P[i].
- **`broadcast ch(m)`** is not indivisible.

Logical Clocks and Event Ordering

- A **logical clock** is a simple integer counter that is incremented when events occur.
- Only communication actions, **send** (including **broadcast**) and **receive**, are the significant *events* in a distributed program.
- The "***happens before***" relationship: a **send** event *happened before* the corresponding **receive** event.

Logical Clock Update Rules

- Let each process have a logical clock and assume that every message contains a timestamp. Let *lc* be a logical clock in process *A*.

1. When *A* sends (or broadcast) a message, it sets the time stamp of this message to the current value of *lc*, then increments *lc* by 1.
2. When *A* receives a message with time stamp *ts* from any process, say *B*, process *A* sets *lc* to be max(*ts+1, lc*), then increments *lc* by 1.

Properties of Logical Clocks:

- Every message sent by *A* will have a different timestamp with an increasing value in the order of sending.
- Upon receiving a message carrying a timestamp by *A*, *ts*, the message subsequently sent by *A* will have a timestamp larger than *ts*.
- If each process has a unique identity, then a total ordering between all events results from using the smaller identity to break any tie of two equal timestamps .

A Partial Ordering on the Causally Related Events

- The logical clock induces a partial ordering on the causally related events in a program.
- If two events having the same *lc* value, we can use the process ids to break this tie.

Distributed Semaphores

- The processes that share a semaphore, *s*, must cooperatively maintain the semaphore invariant of *s*:
  DSEM: *(# of V operations) ≥ (# P operations)*
- ***Data structures***
  - A message: sender's id, operation (P or V), timestamp ts.
  - Every process maintains a logical clock, *lc*.
  - Every process maintains a message queue *mq*.

    mq stores messages in the ascending order of messages' timestamp values. (Senders' ids can be used to break tie.)

- ***Communication Properties***
  - To execute a P or V operation, a process broadcasts a message (to every participating process including itself).
  - Two messages broadcast by two processes might be received by others in different orders. A message with **smaller** *ts* might be received **after** another message with a **larger** *ts* is received .
  - Different messages broadcast by a process will be received by every process in the original sending order. This is because we have perfect channels.

- ***<u>Fully acknowledged</u>***

  Let message *m* carries timestamp *ts*.

  - Once a message with timestamp *ts* coming from process *A* is received, the timestamp of no message from process *A* will be smaller than *ts*.
  - Once the process has received a message, ***m***, with a timestamp larger than *ts* from ***every*** other process, this process will never receive a message with a timestamp smaller than *ts*. At this point, message ***m*** is said to be ***fully acknowledged***.
  - Once ***m*** is fully acknowledged, then every other message in front of it in *mq* will also be fully acknowledged, since they all have a smaller *ts*.

- ***<u>Program</u> (Fig. 9.13)***
  - Every process has a helper process.
  - Every helper has a local variable *sem* to keep track of the semaphore.
  - Both regular and helper processes maintain their own *lc*.
  - Whenever a process receives a P or V message, it will broadcast an ACK message.
  - In the program,

  nV -- number of fully acknowledged V messages

  nP -- number of fully acknowledged P messages

  mq -- totally ordered queue (by *ts*) of P and V messages

9.5 Broadcast Algorithms

```
type kind = enum(reqP, reqV, VOP, POP, ACK);
chan semop[n](int sender; kind k; int timestamp);
chan go[n](int timestamp);

process User[i = 0 to n-1] {
  int lc = 0, ts;
  ...
  # ask my helper to do V(s)
  send semop[i](i, reqV, lc); lc = lc+1;
  ...
  # ask my helper to do P(s), then wait for permission
  send semop[i](i, reqP, lc); lc = lc+1;
  receive go[i](ts); lc = max(lc, ts+1); lc = lc+1;
}
```

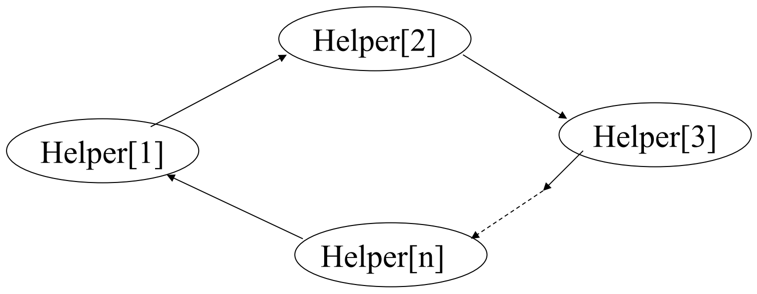
```
process Helper[i = 0 to n-1] {
  queue mq = new queue(int, kind, int);   # message queue
  int lc = 0, s = 0;          # logical clock and semaphore
  int sender, ts; kind k;     # values in received messages
  while (true) {    # loop invariant DSEM
    receive semop[i](sender, k, ts);
    lc = max(lc, ts+1); lc = lc+1;
    if (k == reqP)
      { broadcast semop(i, POP, lc); lc = lc+1; }
    else if (k == reqV)
      { broadcast semop(i, VOP, lc); lc = lc+1; }
    else if (k == POP or k == VOP) {
      insert (sender, k, ts) at appropriate place in mq;
      broadcast semop(i, ACK, lc); lc = lc+1;
    }
    else {  # k == ACK
      record that another ACK has been seen;
      for (all fully acknowledged VOP messages in mq)
        { remove the message from mq; s = s+1; }
      for (all fully acknowledged POP messages in mq st s > 0) {
        remove the message from mq; s = s-1;
        if (sender == i)     # my user's P request
          { send go[i](lc); lc = lc+1; }
      }
    }
  }
}
```

- ***Application considerations***
  - Even though the processes might be at different stages in handling P and V messages, each one will handle fully acknowledged message in the same order.
  - The broadcast algorithms do not scale well to large systems.

9.6 Token-Passing Algorithms

Distributed Mutex

- The critical section problem arises whenever a resource is shared.
- Solutions: monitors, distributed semaphores, logical token rings, ...



9.6 Token-Passing Algorithms

- Every regular process has a help process.
- One token circulates in the ring formed by helper processes. This token cannot be lost.
- The ring invariant DMUTEX:

  User[i] is in its CS $\Rightarrow$ **Helper[i]** has token

  $\wedge$ there is exactly only one token

```
chan token[1:n](), enter[1:n](), go[1:n](), exit[1:n]();

process Helper[i = 1 to n] {
  while (true) {   # loop invariant DMUTEX
    receive token[i]();          # wait for token
    if (not empty(enter[i])) {   # does user want in?
      receive enter[i]();          # accept enter msg
      send go[i]();                # give permission
      receive exit[i]();           # wait for exit
    }
    send token[i%n + 1]();       # pass token on
  }
}

process User[i = 1 to n] {
  while (true) {
    send enter[i]();           # entry protocol
    receive go[i]();
    critical section;
    send exit[i]();            # exit protocol
    non-critical section;
  }
}
```

Termination Detection in a Ring

- Challenge: no visible global state, messages may be in transit between processes.
- Assumptions
  - All communication between processes goes around a ring only.
  - P[i] receives messages from ch[i] only.
  - P[i] sends messages to ch[(i mod n) + 1] only.
  - The states of regular processes are:

    idle -- terminated or blocked by a **receive**

    active -- anything else
  - After receiving a message, an idle process becomes active.
  - A process will not stop to receive a token until it becomes idle

- Termination conditions
  - DTERM: every process is idle $\wedge$ no messages are in transit
- Termination detection
  - A process holding a token passes the token on when it becomes idle.
  - A terminated process will keep passing token on and ignore any regular messages.
  - Upon receiving a token, a process send the token to its neighbor right away.

9.6 Token-Passing Algorithms

- **Decision criterion - in a Ring**

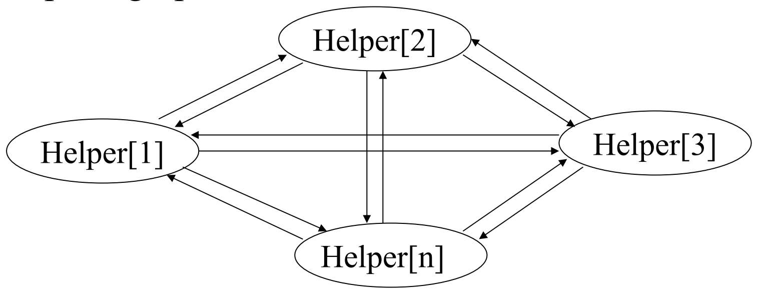
  Suppose P[1] initiates the termination-detection algorithm by passing a token to P[2]. When the token gets back to P[1], the computation has terminated, if P[1] has remained continuously idle since it first passed the token to P[2].

- Program (Fig. 9.16)
  - Initially all processes are active
  - Associate a value with the token indicating how many channels are empty if P[1] is still idle.

```
Global invariant RING:
    T[1] is blue ⇒ ( T[1] ... T[token+1] are blue ∧
                     ch[2] ... ch[token%n + 1] are empty )

actions of T[1] when it first becomes idle:
    color[1] = blue; token = 0; send ch[2](token);

actions of T[2], ..., T[n] upon receiving a regular message:
    color[i] = red;

actions of T[2], ..., T[n] upon receiving the token:
    color[i] = blue; token++; send ch[i%n + 1](token);

actions of T[1] upon receiving the token:
    if (color[1] == blue)
      announce termination and halt;
    color[1] = blue; token = 0; send ch[2](token);
```

Termination detection in a complete graph

- Having the token traversed every node is not sufficient.
- Let the token traverses *every edge* of the graph. If every process has remained continuously idle since it first saw the token, then the computation is terminated.
- Complete graph



- Program (Fig. 9.17)
  - Let *c* be a circuit covering the graph. *nc* = length(*c*).
  - Since *c* may goes through a node multiple times, each process keeps track of the order in which its outgoing edges occur in *c*.
  - Upon receiving a token, a process sends it out over the next edge in *c*.
  - The token carries a value indicating the number of times that the token has been passed by idle processes.
  - If a process is red when it receives a token, the process colors itself blue, and set the token value to be 0, then pass the token over the next edge.
  - If the process is blue when it receives the token, the process increments the token value by 1, then pass the token over the next edge .

```
Global invariant GRAPH:
    token has value V  ⇒
          ( the last V channels in cycle C were empty ∧
          the last V processes to receive the token were blue )

actions of T[i] upon receiving a regular message:
    color[i] = red;

actions of T[i] upon receiving the token:
    if (token == nc)
        announce termination and halt;
    if (color[i] == red)
        { color[i] = blue; token = 0; }
    else
        token++;
    set j to index of channel for next edge in cycle C;
    send ch[j](token);
```

Figure 9.18 Termination detection in a complete graph.

9.6 Token-Passing Algorithms

- **Decision criterion - in a Complete Graph**
  - Once the token value reaches *nc*, the computation is known to have terminated.
  - When did the computation actually terminate? It terminated when the token started its last circuit around the graph. However, no process could possibly know this.
  - The token has to circulate a minimum of two times around the cycle after any activity of the computation. (The first round turns processes to blue; the second verifies that they have remained blue.)

9.7 Replicated Servers

- Why?
  - To increase accessibility, to speed up.
- Three ways to maintain replicated files:
  - manually, periodically backup (copy) files;
  - automatically backup (copy) files;
  - **transparent replication provided by the file system**
- File consistency -- a special case of the readers/writers problem
  - One client process interacts with one file server only.
  - The file servers interact with each other to produce a single-copy illusion.

- **Solution 1** -- By the distributed solution to the critical section problem (mutex enforced with a token ring).

read:	the server reads the local copy of the file.
write:	the server updates all copies of the file.
Drawbacks:	more restrict than necessary

- **Solution 2** -- Let token to be a counter. Concurrent reading is allowed.
- *initialize:* token value = *ns*, (# of servers, # of readers allowed)
- *read:* when a client process opens a file for reading, its server waits for the token and decrements it by 1, then sends the token to the next server, then handles the client process.

  After the client process closes the file, its server increments the token by 1 when the token arrives, then sends the token to the next server.

  write: when a client process opens a file for writing, its server waits for the token to have value *ns* and holds the token while handling read/write requests.

  After the client process closes the file, its server updates all the copies, then recovers the token value to be *ns* and sends the token to the next server.
- If readers keep coming, a writer will never be able to write.

- **Solution 3** -- Use *ns* tokens
- *initialize:* each server has one token.
- *read:* when a client process opens a file for reading, its server must obtain one token.

  write: when a client process opens a file for writing, its server must obtain ns tokens by doing the following:
  - the server sends a message to every other server requesting a token;
  - the server collects tokens until it obtains *ns* tokens;
  - once it obtains ns tokens, the server holds these *ns* tokens while handling the r/w request.

passing tokens: upon receiving a request for a token the server, *s*, will

- do nothing, if *s* does not have a token or if *s* is collecting tokens for a write request that happened before the timestamp carried by this token-request message;
- give up its token if *s* is not serving any request;
- give up its token(s) if *s* is collecting tokens for a write request that happened after the timestamp carried by this token-request message.

closing: After the client process closes the file, its server updates all the copies, then recovers *ns* tokens and sends the tokens to the next server.

Advantage: A server can continue to process write requests until some other servers request tokens.

- **Solution 4 -- weighted voting**

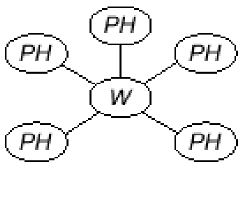
  Let *rw* be the read weight and *ww* be the write weight satisfying

  $ww > ns/2$

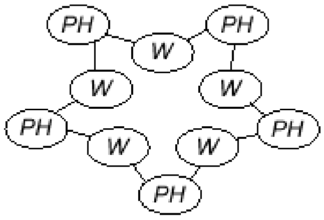
  $(rw + ww) > ns$

  - Only *ww* copies of the file are necessary to be updated when a write is processed. However, it is necessary to read *rw* copies.
  - Every write action sets a timestamp on the copy it writes.
  - By reading, a reader must look for the latest version.
  - Inherently fault-tolerant -- as long as max(*rw, ww*) copies of the file are accessible, the services can be provided

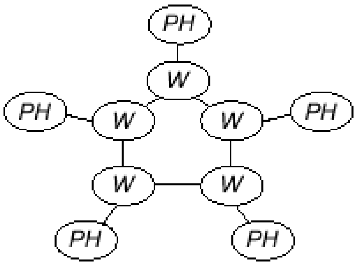
- Solutions to the Dinning Philosopher Problem



(a) Centralized



(b) Distributed



(c) Decentralized

– A Centralized solution

```
module Table
  op getforks(int), relforks(int);
body
  process Waiter {
    bool eating[5] = ([5] false);
    while (true)
      in getforks(i) and not (eating[left(i)] and
           not eating[right(i)] -> eating[i] = true;
      [] relforks(i) ->
            eating[i] = false;
      ni
  }
end Table

process Philosopher[i = 0 to 4] {
  while (true) {
    call getforks(i);
    eat;
    call relforks(i);
    think;
  }
}
```

– A distributed dinning philosopher solution

```
module Waiter[5]
  op getforks(), relforks();
body
  process the_waiter {
    while (true) {
      receive getforks();
      receive relforks();
    }
  }
end Waiter

process Philosopher[i = 0 to 4] {
  int first = i, second = i+1;
  if (i == 4) {
    first = 0; second = 4; }
  while (true) {
    call Waiter[first].getforks();
    call Waiter[second].getforks();
    eat;
    send Waiter[first].relforks();
    send Waiter[second].relforks();
    think;
  }
}
```

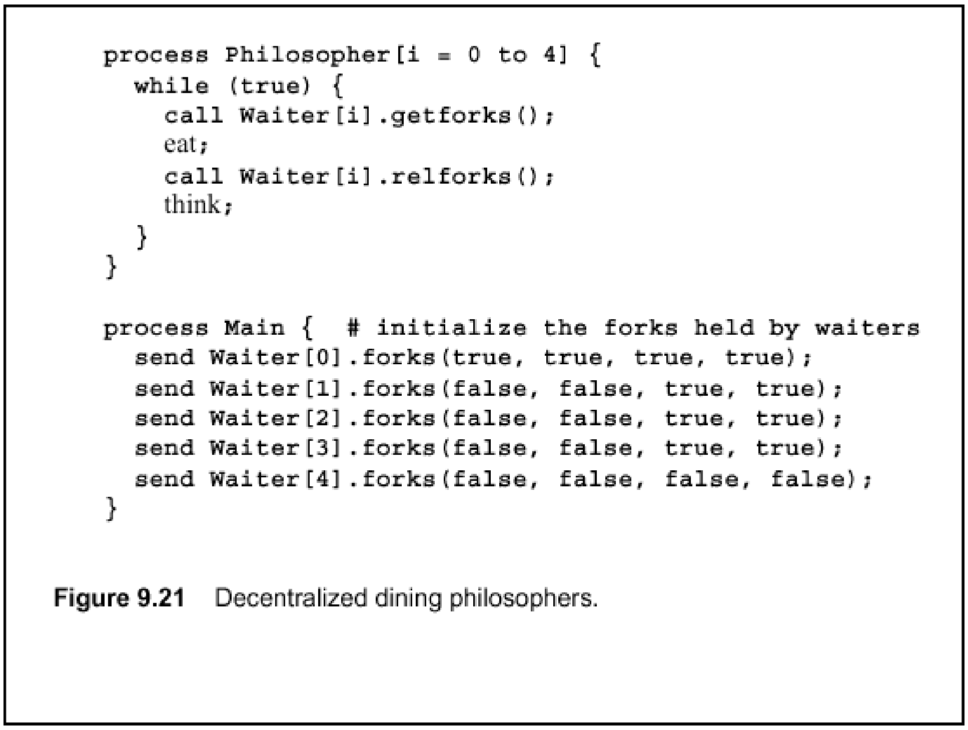
9.7 Replicated Servers

– A decentralized dinning philosopher solution

```
module Waiter[t = 0 to 4]
  op getforks(int), relforks(int);  # for philosophers
  op needL(), needR(),              # for waiters
     passL(), passR();
  op forks(bool,bool,bool,bool);  # for initialization
body
  op hungry(), eat();      # local operations
  bool haveL, dirtyL, haveR, dirtyR;  # status of forks
  int left = (t-1) % 5 ;       # left neighbor
  int right = (t+1) % 5;       # right neighbor

  proc getforks() {
    send hungry();  # tell waiter philosopher is hungry
    receive eat();  # wait for permission to eat
  }
```

```
process the_waiter {
  receive forks(haveL, dirtyL, haveR, dirtyR);
  while (true) {
    in hungry() ->
        # ask for forks I don't have
        if (!haveR) send Waiter[right].needL();
        if (!haveL) send Waiter[left].needR();
        # wait until I have both forks
        while (!haveL or !haveR)
          in passR() ->
              haveR = true; dirtyR = false;
          [] passL() ->
              haveL = true; dirtyL = false;
          [] needR() st dirtyR ->
              haveR = false; dirtyR = false;
              send Waiter[right].passL();
              send Waiter[right].needL()
          [] needL() st dirtyL ->
              haveL = false; dirtyL = false;
              send Waiter[left].passR();
              send Waiter[left].needR();
          ni
        # let philosopher eat, then wait for release
        send eat(); dirtyL = true; dirtyR = true;
        receive relforks();
```

```
        [] needR() ->
            # neighbor needs my right fork (its left)
            haveR = false; dirtyR = false;
            send Waiter[right].passL();
        [] needL() ->
            # neighbor needs my left fork (its right)
            haveL = false; dirtyL = false;
            send Waiter[left].passR();
        ni
      }
    }
  end Waiter
```



```
process Philosopher[i = 0 to 4] {
  while (true) {
    call Waiter[i].getforks();
    eat;
    call Waiter[i].relforks();
    think;
  }
}

process Main {  # initialize the forks held by waiters
  send Waiter[0].forks(true, true, true, true);
  send Waiter[1].forks(false, false, true, true);
  send Waiter[2].forks(false, false, true, true);
  send Waiter[3].forks(false, false, true, true);
  send Waiter[4].forks(false, false, false, false);
}
```

Figure 9.21 Decentralized dining philosophers.