Asynchronous Computations

Thoai Nam

High Performance Computing Lab (HPC Lab)

Faculty of Computer Science and Technology

HCMC University of Technology

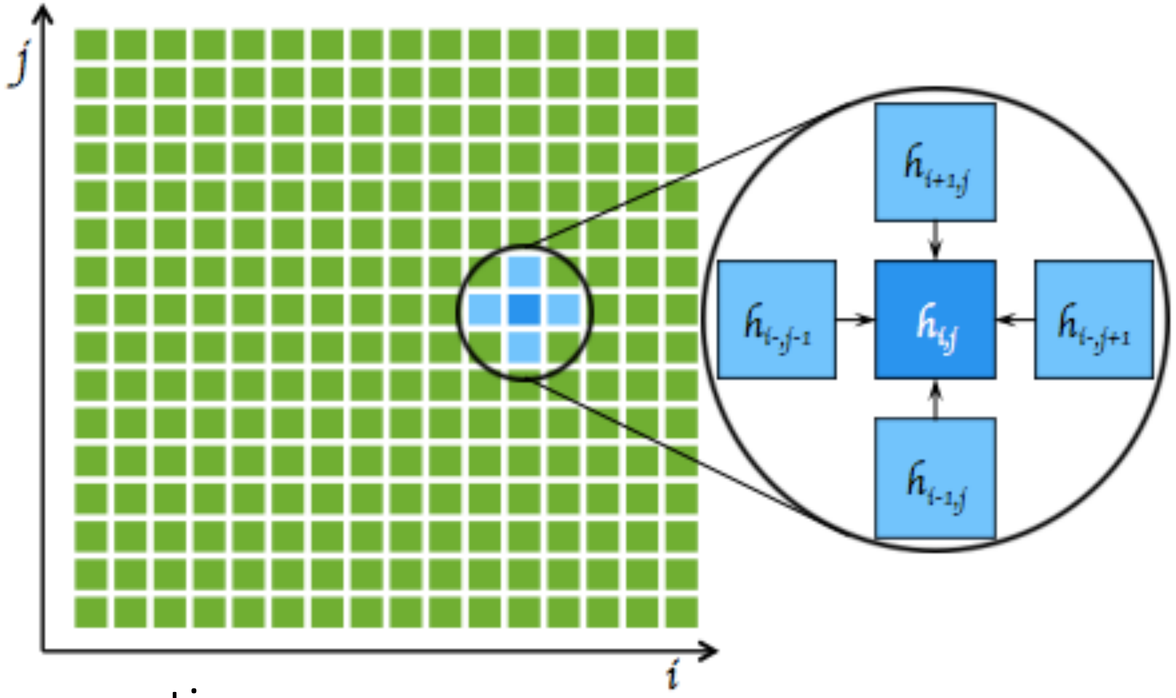
Asynchronous computations

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

- 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.

Heat distribution problem
(Locally synchronous computation)

- An area has known temperatures along each of its edges
- Find the temperature distribution within
- Divide area into fine mesh of points $h_{i,j}$. Temperature at an inside point taken to be average of temperatures of four neighboring points. Convenient to describe edges by points.



  - Temperature of each point by iterating the equation:

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

($0 < i < n$, $0 < j < n$) for a fixed number of iterations or until the difference between iterations less than some very small amount.

Sequential algorithms

```
Seq_heat_distribution_ver1 () {
  do {
    for (k=0; k<Max_loop; k++) { // Lặp đến Max_loop
  // Tính toán giá trị nhiệt mới tại bước k, không tính ở biên
      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]);
  // Cập nhật giá trị nhiệt mới tại bước k và h[i][j]
      for (i=1; i<n; i++)
        for (j=1; j<n; j++)
          h[i][j] = g[i][j];
  // Kiểm tra điều kiện kết thúc
      continue = false;
      for (i=1; i<n; i++)
        for (j=1; j<n; j++)
          if !converged(i, j) {
            continue = true;
            break;
          }
  // Dừng khi đạt điều kiện kết thúc hoặc lặp đủ Max_loop bước
  } while ((continue == true) && (k < (Max_loop-1)))
}
```

```
Seq_heat_distribution_ver2 () {
  do {
    for (k=0; k<Max_loop; k++) { // Lặp đến Max_loop
  // Tính toán giá trị nhiệt mới tại bước k, không tính ở biên
      for (i=1; i<n; i++)
        for (j=1; j<n; j++)
          h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                                  h[i][j-1] + h[i][j+1]);
  // Kiểm tra điều kiện kết thúc
      continue = false;
      for (i=1; i<n; i++)
        for (j=1; j<n; j++)
          if !converged(i, j) {
            continue = true;
            break;
          }
  // Dừng khi đạt điều kiện kết thúc hoặc lặp đủ Max_loop bước
  } while ((continue == true) && (k < (Max_loop-1)))
}
```

Parallel algorithm

```
// Lặp đến Max_loop
for (k=0; k<Max_loop; k++) {
  h = 0.25 * (l + r + d + u);
// Send() ở chế độ
// không bị chặn (non-blocking)
  Send(&h, P_{i-1,j});
  Send(&h, P_{i+1,j});
  Send(&h, P_{i,j-1});
  Send(&h, P_{i,j+1});
// Recv() ở chế độ hay đồng bộ
// (synchronous) bị chặn (blocking)
  Recv(&l, P_{i-1,j});
  Recv(&r, P_{i+1,j});
  Recv(&d, P_{i,j-1});
  Recv(&u, P_{i,j+1});
}
```

Local barrier

```
Parrallel_heat_distribution () {
  do {
    for (k=0; k<Max_loop; k++) { // Lặp đến Max_loop
    // Tính toán giá trị nhiệt mới tại bước k, không tính ở biên
      forall (i=1; i<n; i++)
        forall (j=1; j<n; j++)
          h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                            h[i][j-1] + h[i][j+1]);
    // Kiểm tra điều kiện kết thúc
      continue = false;
      for (i=1; i<n; i++)
        for (j=1; j<n; j++)
          if !converged(i, j) {
            continue = true;
            break;
          }
    // Dừng khi đạt điều kiện kết thúc hoặc lặp đủ Max_loop bước
  } while ((continue == true) && (k < (Max_loop-1)))
}
```

Overhead → Barrier

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

Asynchronous computations

- 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).

Overall parallel code

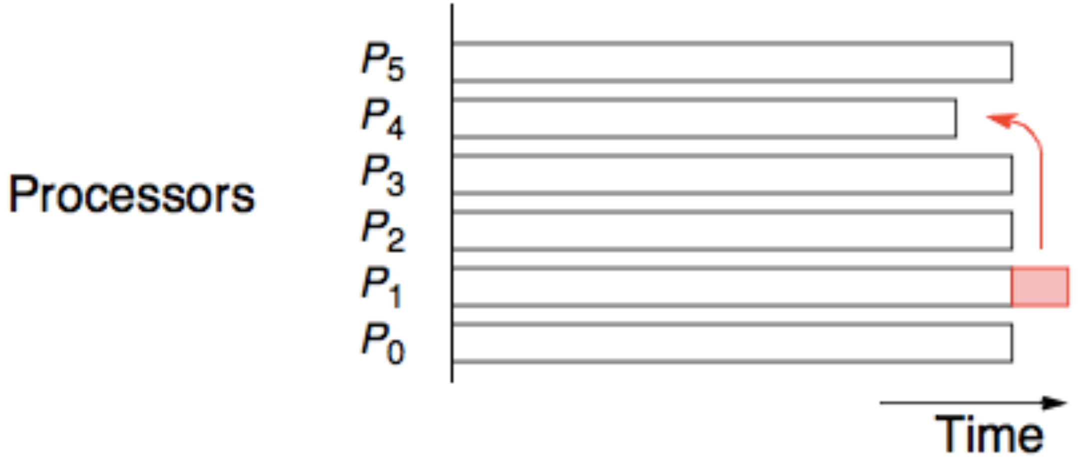
- Each process allowed to perform `s` iterations before being synchronized and also to update the array as it goes. At `s` iterations, maximum divergence recorded. Convergence is checked then.
- The actual iteration corresponding to the elements of the array being used at any time may be from an earlier 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.

Load balancing and Termination detection

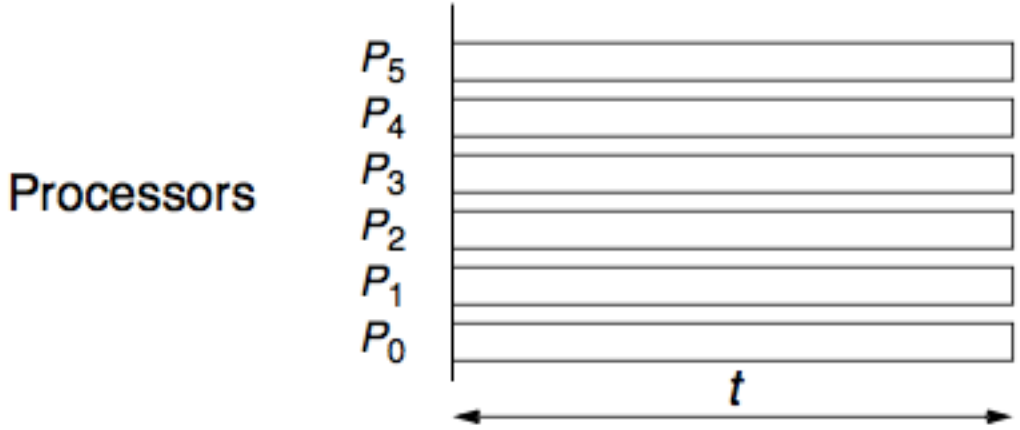
- Load balancing – used to distribute computations fairly across processors in order to obtain the highest possible execution speed.

- 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 (1)

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.

Static load balancing (2)

- 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

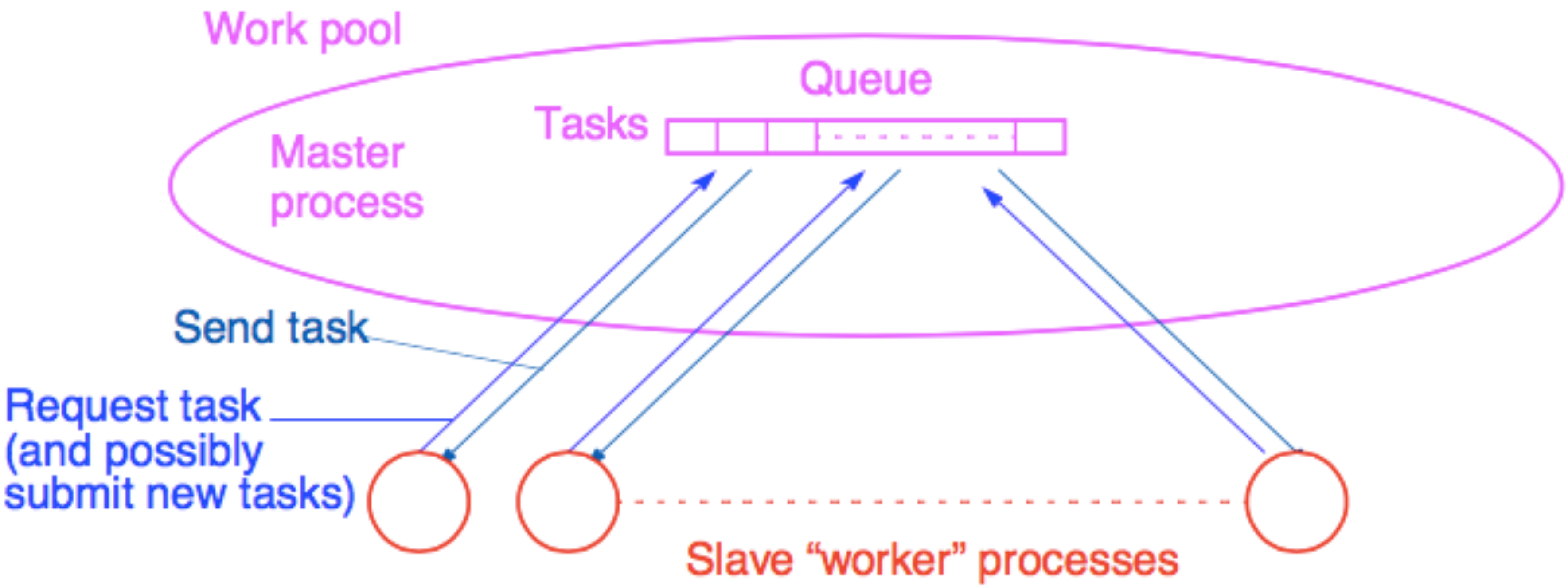
Can be classified as:

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

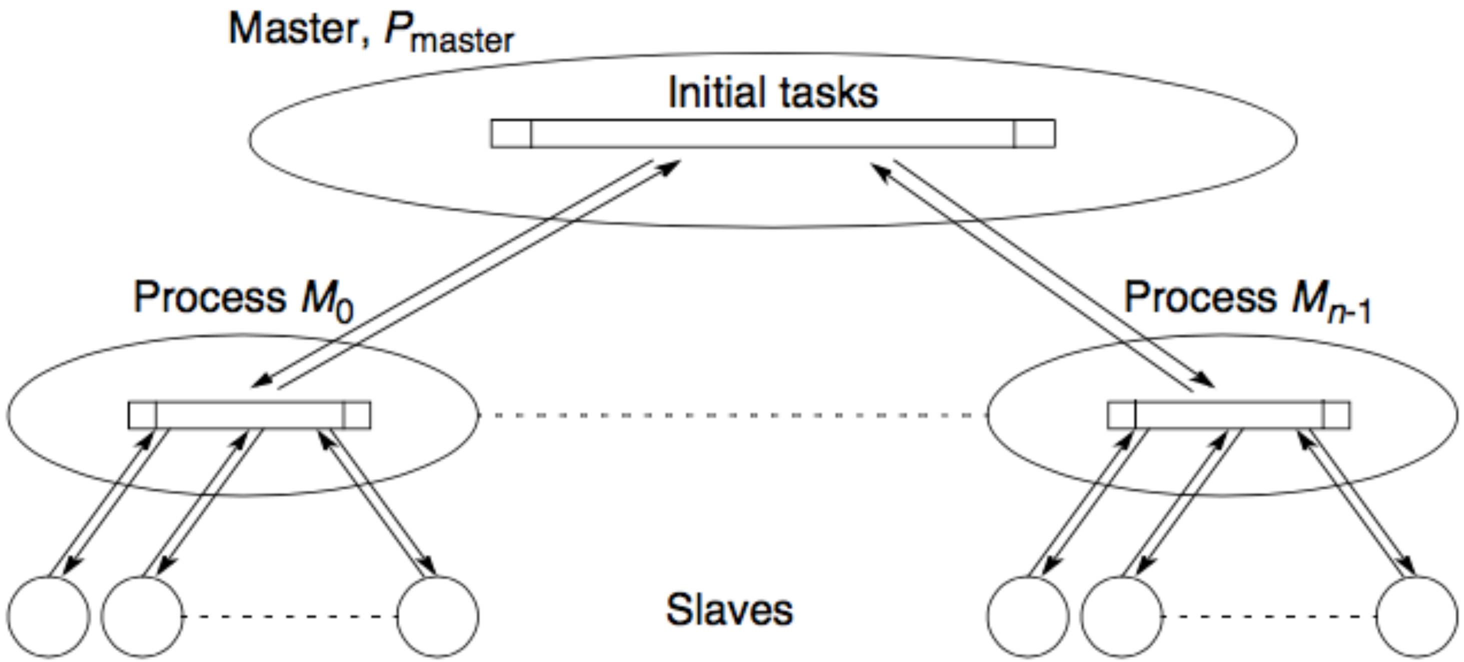
Computation terminates when:

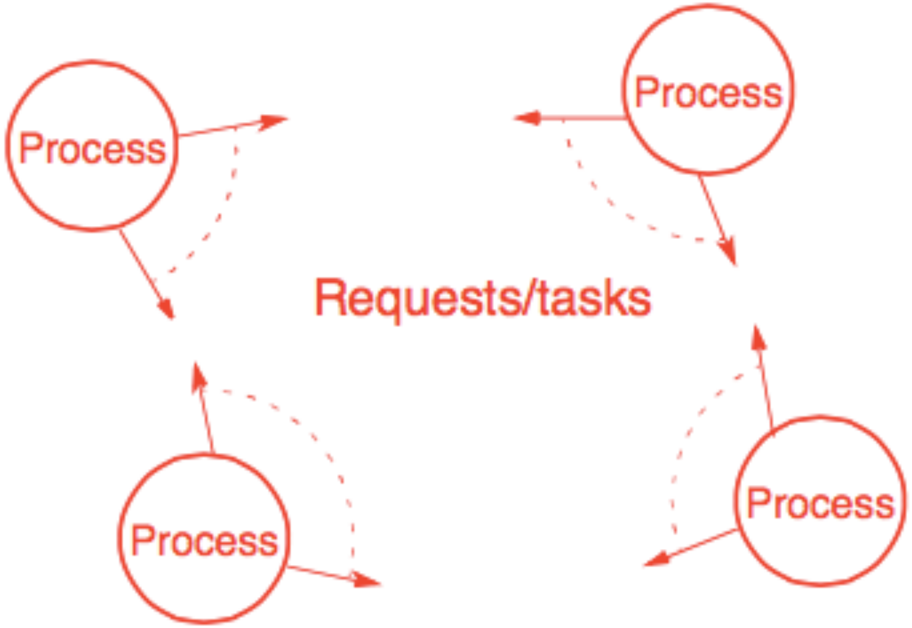
- The task queue is empty and
- Every process has made a request for another task without any new tasks being generated

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

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.

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.

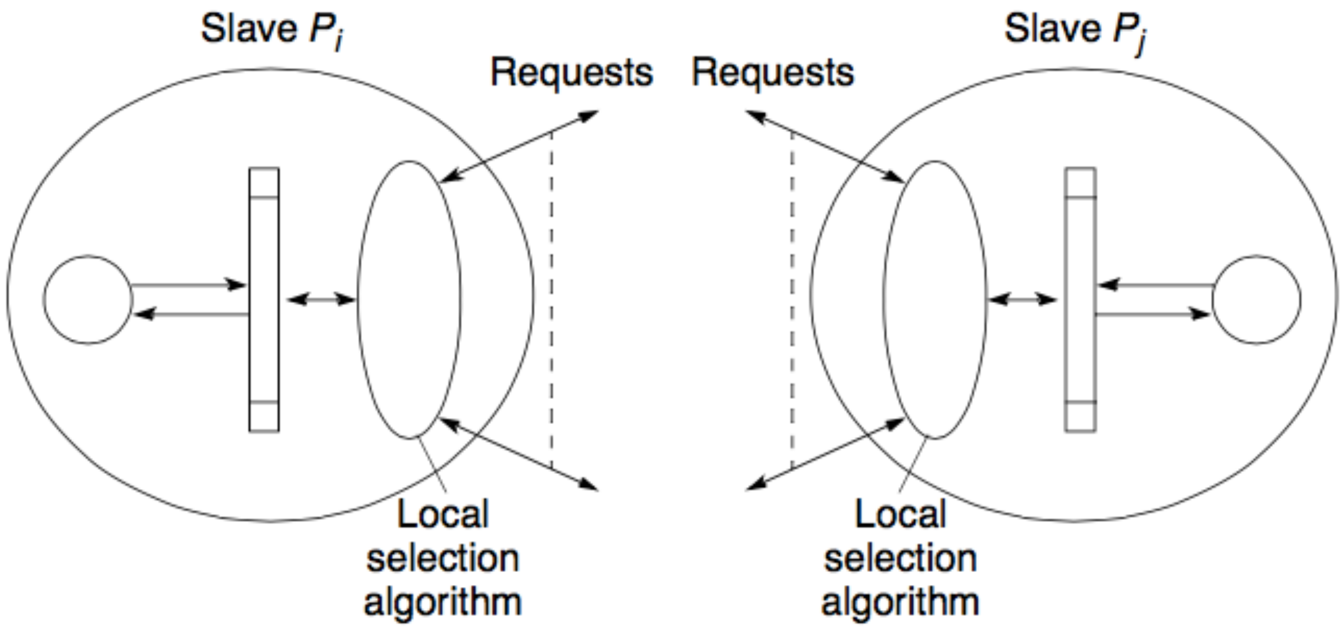
- Another option is to have a mixture of both methods. Unfortunately, it can be expensive to determine process loads.
- In very heavy system loads, load balancing can also be difficult to achieve because of the lack of available processes.

Decentralized selection algorithm requesting tasks between slaves

Algorithms for selecting a process:

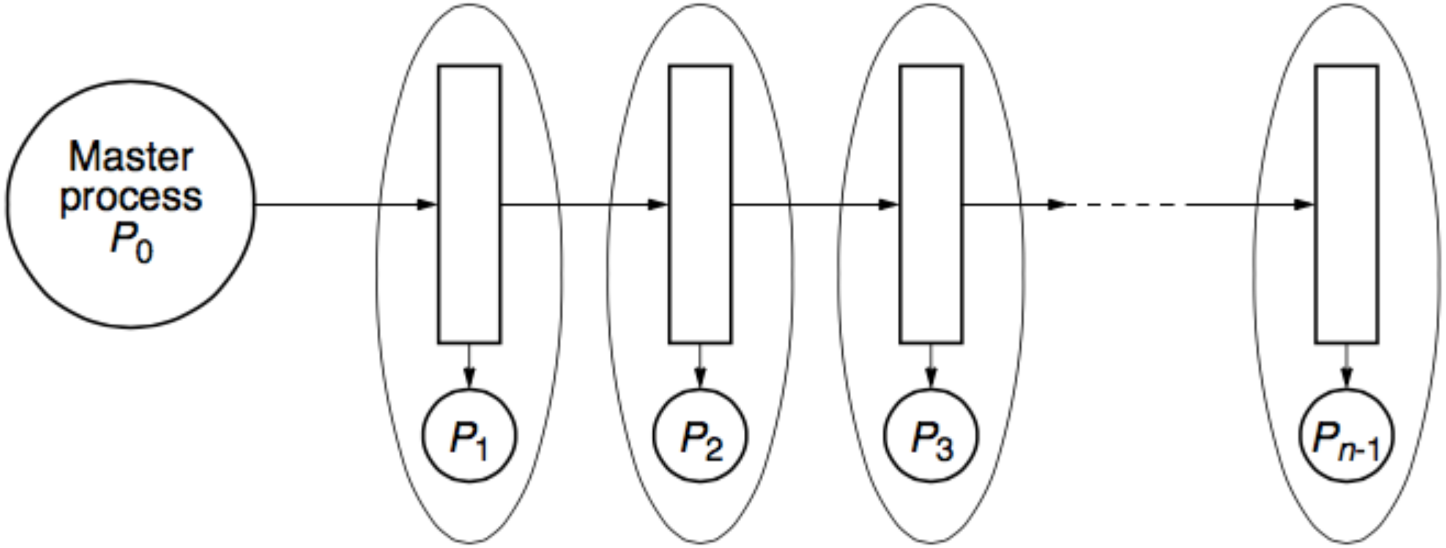
- Round robin algorithm – process P_i requests tasks from process P_x , where x is given by a counter that is incremented after each request, using modulo n arithmetic (n processes), excluding $x = i$.

- 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 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.

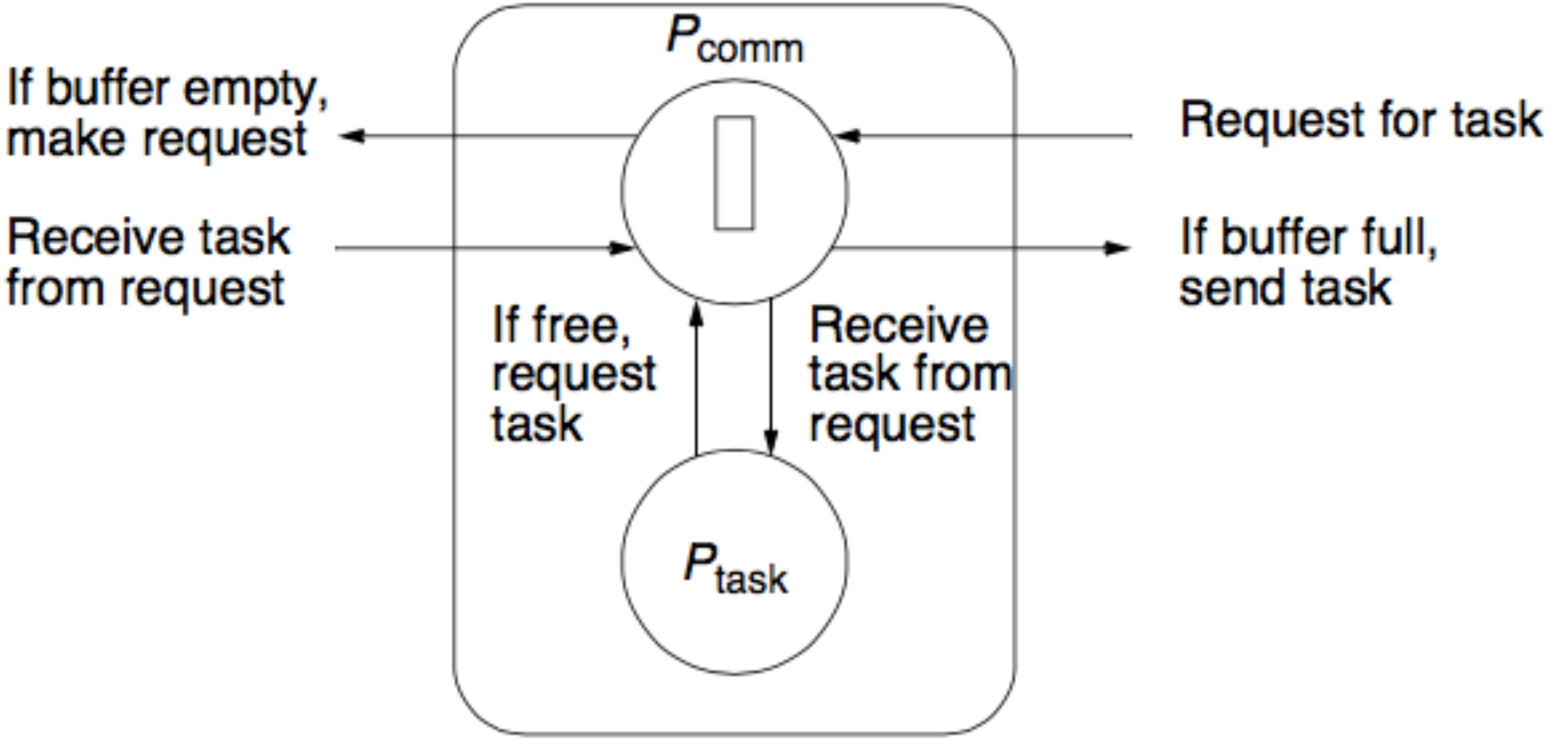


- Then the tasks to the left shuffle down the queue so that the space held by the task is filled. A new task is inserted into the left side end of the queue.
- Eventually, all processes will have a task and the queue is filled with new tasks.
- High-priority or larger tasks could be placed in the queue first.

Shifting actions

Could be orchestrated by using messages between adjacent processes:

- For left and right communication
- For the current task.



Using time sharing between communication and computation

Master process (P_0)

```
for (i = 0; i < no_tasks; i++) {
    recv(P_1, request_tag); // request for task
    send(&task(i), P_1, task_tag); // send tasks into queue
}
recv(P_1, request_tag); // request for task
send(&empty, P_1, task_tag); // end of tasks
```

nonblocking

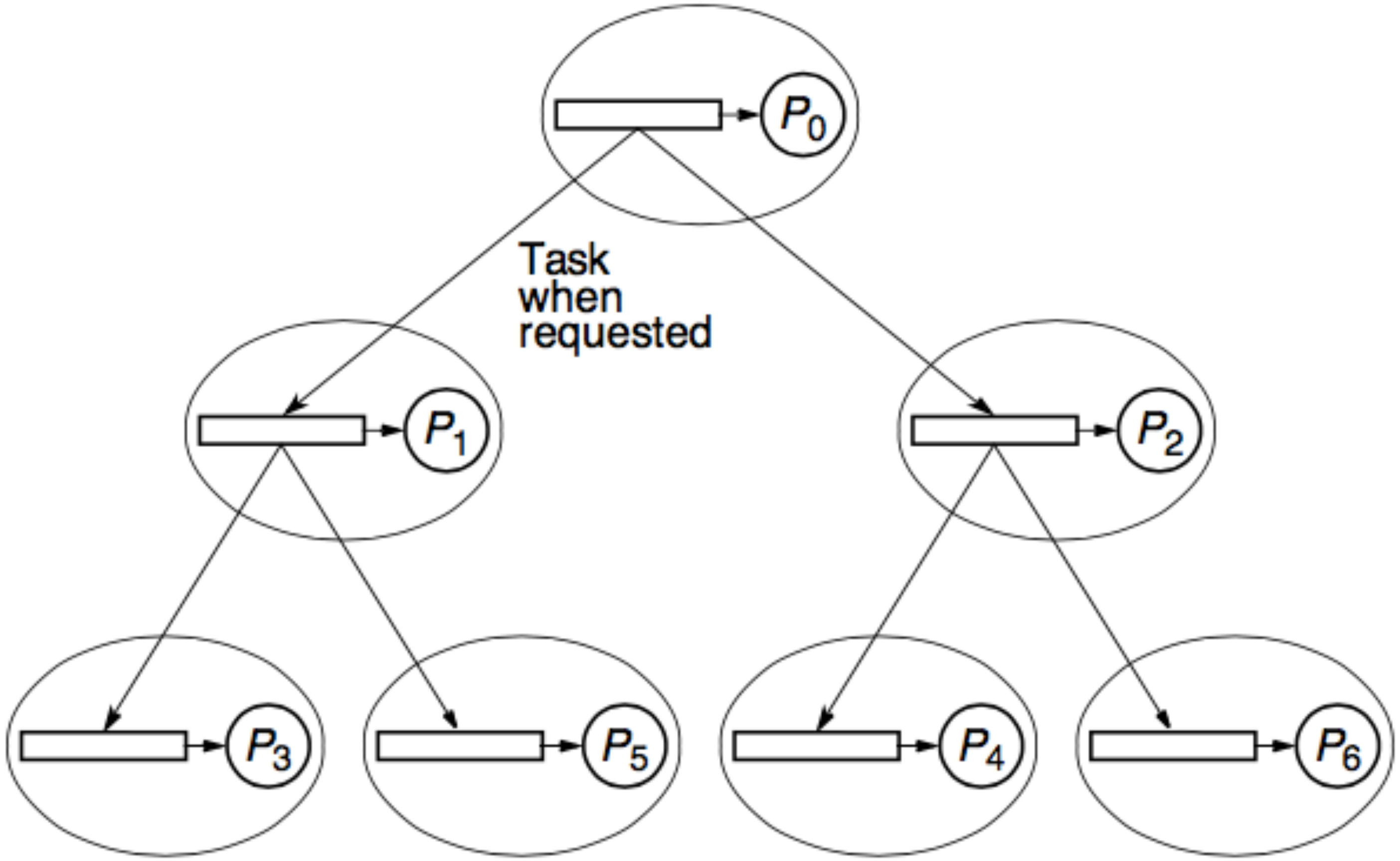
- Nonblocking receive, MPI_Irecv(), returns a request "handle," which is used in subsequent completion routines to wait for the message or to establish whether the message has actually been received at that point (MPI_Wait() and MPI_Test(), respectively).
- In effect, the nonblocking receive, MPI_Irecv(), posts a request for message and returns immediately.

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.
}
```

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

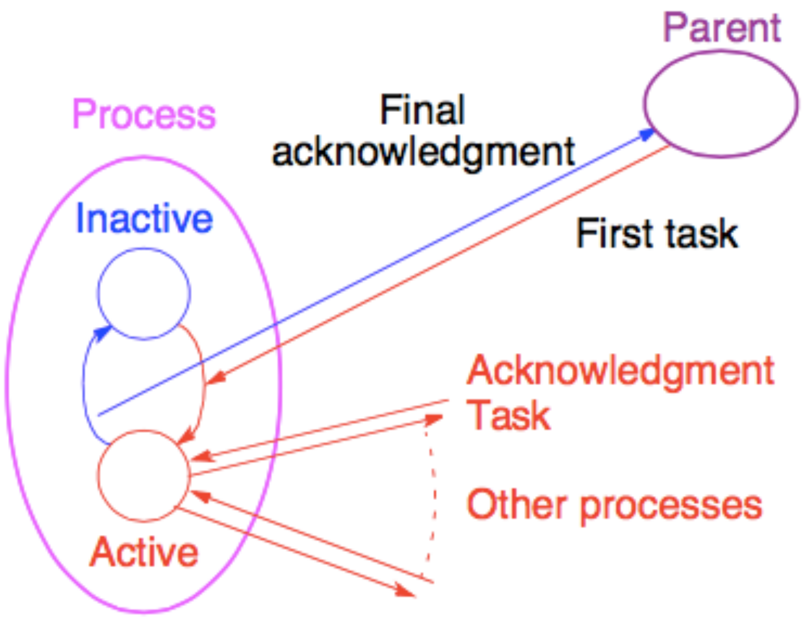
Termination Conditions

- 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.

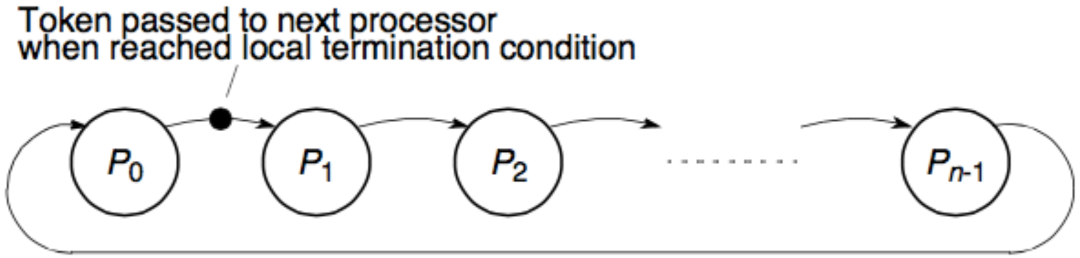
One very general distributed termination algorithm

- Each process in one of two states:
  - Inactive - without any task to perform
  - 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.

Ring termination algorithms

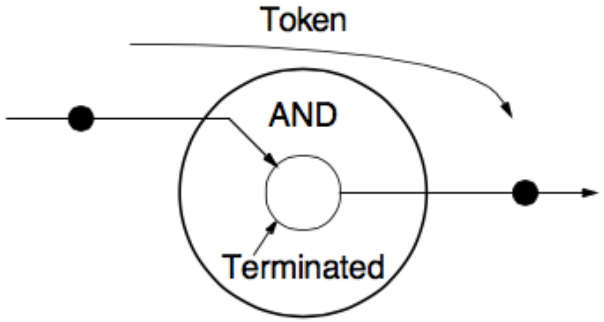


Single-pass ring termination algorithm

- When P_0 has terminated, it generates a token that is passed to P_1
- 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
- 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.

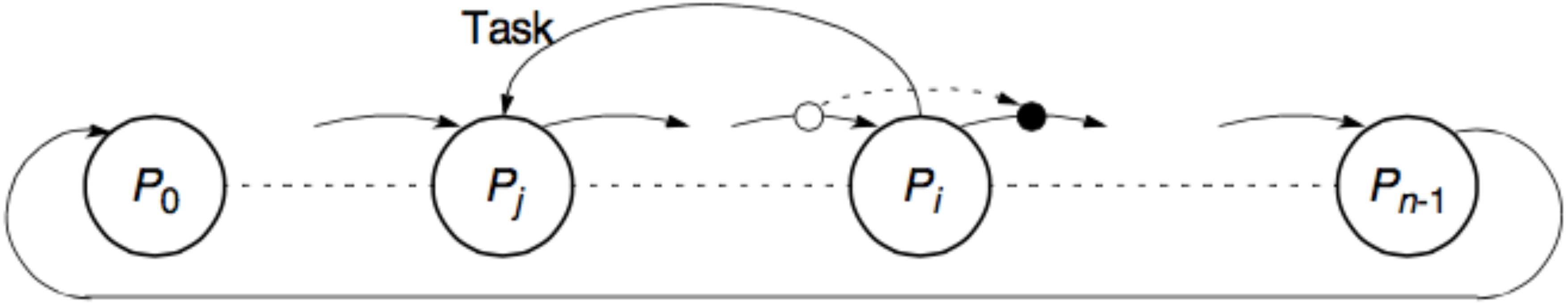
✓ The algorithm assumes that a process cannot be reactivated after reaching its local termination condition

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



Dual-pass ring termination algorithm (1)

- 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 around ring again.



Dual-pass ring termination algorithm (2)

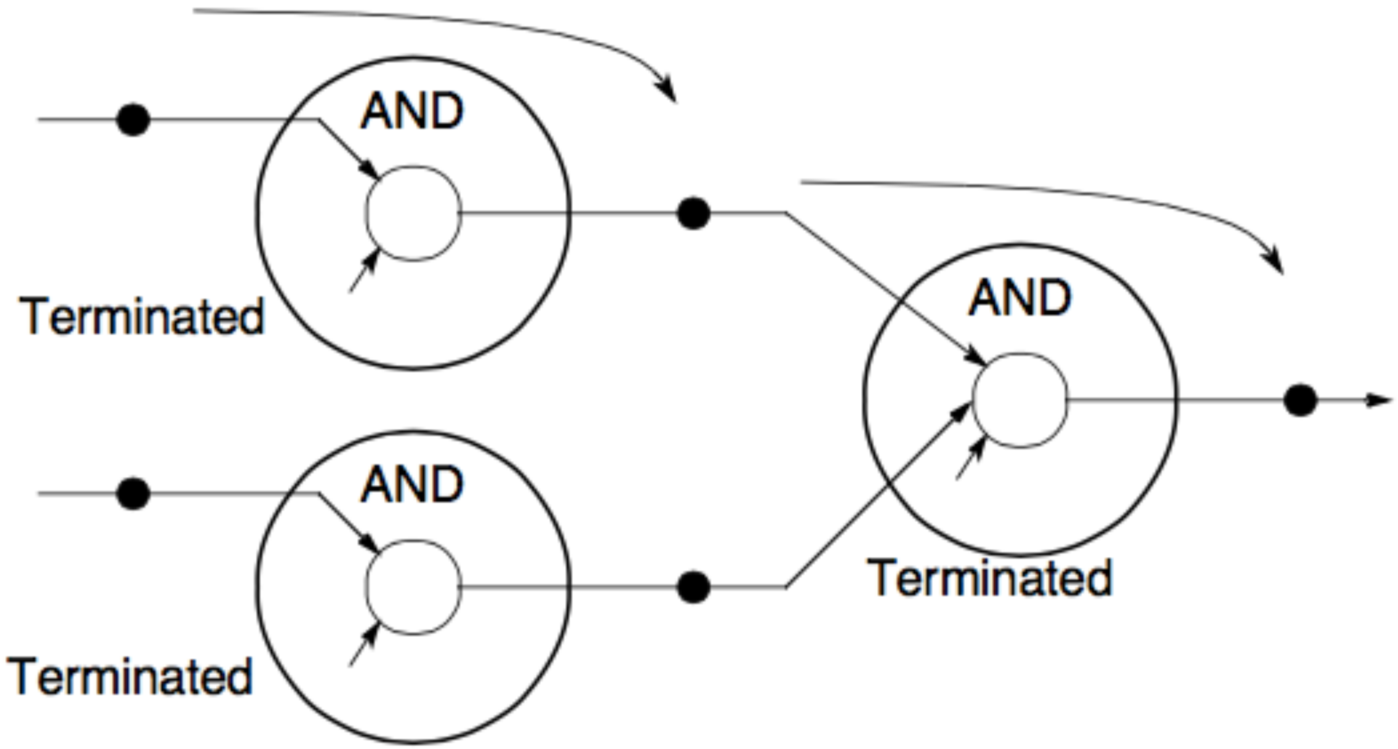
The algorithm is as follows, again starting at P_0:

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.

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.