COMP90025 Parallel and Multicore Computing

Parallel algorithm techniques

Aaron Harwood
with contributions from Tuck Leong Ngun

School of Computing and Information Systems
The University of Melbourne

2019 Semester II

Classes of parallel algorithms

The degree of parallelism admitted by an algorithm can vary, as has been seen for fundamental algorithms such as prefix sum, matrix multiplication and sorting.

Algorithms can be loosely classified as one (or more) of:

- embarrassingly parallel,
- parametric,
- data parallel,
- task parallel,
- loosely synchronous, and
- synchronous.

Geometric transformations of images

An image is a matrix, A, where $a_{i,j}$ (the i-th row and j-column) is the colour of pixel (i, j).

It is sometimes convenient to think of the origin $(0, 0)$ being at the top left of the screen. In general it is semantically easier to write $a_{x,y}$ as the pixel at location (x, y) with $(0, 0)$ being at the bottom left and we will do so in this subject.

Using an $n \times m$ matrix, the number of pixels is $N = nm$.

Geometric transformations are mathematical operations computed for each pixel and in this case we consider only those transformations that can be computed completely independently of all other pixels.

Shifting

Shifting requires changing the coordinates of each pixel by a given amount.

$$x' = x + \Delta x$$
$$y' = y + \Delta y$$

Generally, $a'_{x',y'} = a_{x,y}$.

Scaling

Scaling requires multiplying each coordinate by some factor.

$$x' = xS_x$$
$$y' = yS_y$$

If S_x or S_y are less than one then the image will be reduced in size. If S_x or S_y are greater than one then the image will be enlarged in size.

Rotation

Rotation requires each coordinate to be rotated through an angle θ, in this case about the origin of the coordinate system.

$$x' = x\cos\theta + y\sin\theta$$
$$y' = -x\sin\theta + y\cos\theta$$

Clipping

Clipping eliminates points that are not within some boundaries. Only the points (x', y') such that

$$\begin{aligned} x_l \leq x' \leq x_h \\ y_l \leq y' \leq y_h \end{aligned}$$

are kept, the other points are deleted.

Image smoothing

Each pixel may become a function of surrounding pixels, e.g. the mean:

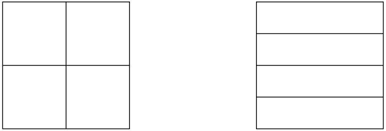
$$x_4' = \frac{x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8}{9}.$$

x_0	x_1	x_2
x_3	x_4	x_5
x_6	x_7	x_8

Partitioning

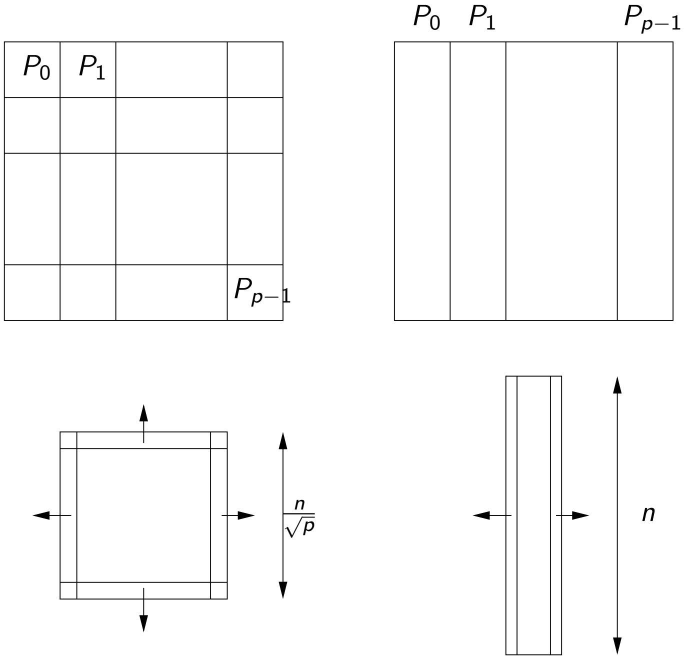
There are many methods for partitioning space. Two basic methods are:

- partitioning into rows,
- partitioning into regions.



If communication between processes does not take place then either method is acceptable.

However, suppose an image operation like smoothing or edge detection requires to be applied some number of times. Then the results of one iteration are required to be communicated to neighboring regions for the next iteration to take place.


P_0
P_1
P_{p-1}
P_0
P_1
P_{p-1}
$\frac{n}{\sqrt{p}}$
n

For square regions:

$$t_{commsq} = 8\left(t_{startup} + \frac{n}{\sqrt{p}} t_{data}\right).$$

Note that we are considering when p is sufficiently large.
For columns:

$$t_{commcol} = 4(t_{startup} + n\, t_{data}).$$

Generally the column partition is better for large startup time and the block partition is better for small startup time. We can ask "when is the square paritioning worse than the column partitioning?":

$$t_{commsq} > t_{commcol}$$

and solve for $t_{startup}$ that maintains this inequality:

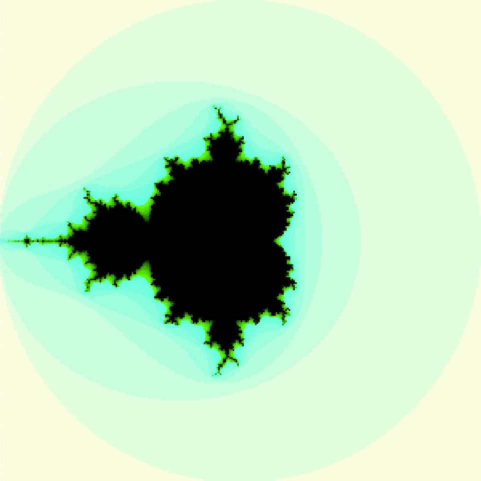
$$t_{startup} > n\left(1 - \frac{2}{\sqrt{p}}\right) t_{data}.$$

Mandelbrot Set

The Mandelbrot Set is an interesting set of points in the complex plane. For a given point, c, in the complex plane, the iteration

$$z_{k+1} = z_k^2 + c$$

is continued until the magnitude of z_{k+1} is greater than 2 or the number of iterations reaches some arbitrary limit. The initial condition is $z_0 = 0$. The magnitude of $z = z_{k+1}$ is $\sqrt{a^2 + b^2}$ where $z = a + bi$.



The image has the real part in the x dimension and the imaginary part in the y dimension. Each dimension ranges from -2 to 2.

The colour of a pixel at location (a, b) is determined by the number of iterations that was required before reaching the termination condition for each $c = a + bi$. Producing fascinating colours requires additional colour computations.

Although each pixel can be computed independently, much like image transformations, the number of iterations required per pixel cannot, in general, be predicted. This means that equally dividing the number of pixels over the available processors will not equally divide the computational work.

The computations required per pixel are not amenable to parallelization.

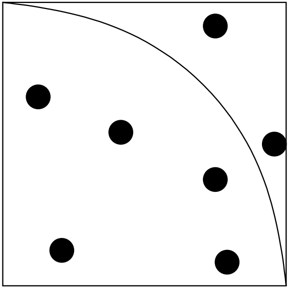
Dynamic Load Balancing

In dynamic load balancing, tasks are allocated to processors during the execution of the program. There are two main classifications:

- Centralised - Single master process holds the task queue and handles the division of tasks
- Hierarchical and decentralised - Some form of distributed load balancing such that multiple processes decide how the tasks are distributed, e.g.
  - "Main master" allocating "mini masters" which collect results from slaves then pass results to the main master (for example many local maxima to find a global maximum)
  - Extension of above: tree structure with leaves as slaves and internal nodes dividing up work
  - Receiver-initiated and Sender-initiated decentralized work distribution: processes act independently and either initiate a "pull" request for work or initiate a "push" request of work. All work starts from an initial process.

Monte Carlo Methods

An interesting problem to calculate the value of π uses Monte Carlo methods. In this problem, darts are thrown at a square dart board that contains a quarter circle.



The darts are randomly located on the dart board.

Let the dart board be the unit square and the location of a dart given by (x, y).

Then the fraction of darts with $x^2 + y^2 \leq 1$ is roughly $\frac{\pi}{4}$ which is the area of the quadrant.

Although all darts can be thrown in parallel, care must be taken to ensure that random numbers are indeed random over all the processes.

For example, using `rand()` at each process may result in the same sequence being produced. This is possible even if different seeds are chosen, since the sequence produced from one seed can overlap with the sequence produced from another seed.

To avoid randomness problems we could generate all random numbers using a single source, such as the master generating all the random numbers. However this doesn't provide good parallelism and there are questions as to whether the randomness of the subsequences is preserved.

A linear congruential generator provides a sequence of pseudo-random numbers using the iterative function:

$$x_{i+1} = (ax_i + c) \mod m.$$

A "good" generator is known to have $a = 16807$, $m = 2^{31} - 1$ (a prime number) and $c = 0$. This generator creates a repeating sequence of $2^{31} - 2$ different numbers.

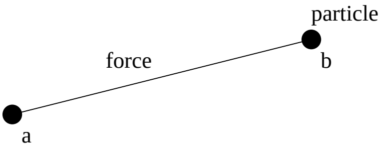
For parallel programs it is known that

$$x_{i+1} = (ax_i + c) \mod m$$
$$x_{i+k} = (Ax_i + C) \mod m$$

where $A = a^k \mod m$, $C = c(a^{k-1} + a^{k-2} + \cdots + a^1 + a^0) \mod m$ and k is a selected step size.
Given p processors, the first p numbers of the sequence are generated sequentially and then each of these numbers is used to generate the next p numbers in parallel.

N-Body Problem

The N-body problem is concerned with forces between “bodies” or particles in space. Each pair of particles generates some force and so, for N particles each particle experiences $N-1$ different forces that total to produce a net force. The net force accelerates the particle.
The force is dependent on the distances between particles and so, as the particles move the forces need to be recalculated.



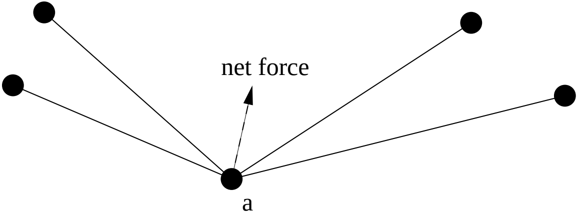
Consider particles that move under the force of gravity:

$$F = \frac{G m_a m_b}{d^2},$$

where F is the force (directed towards the center of the two particles), G is the Gravitational constant, m_a and m_b are the masses of particles a and b, and d is the distance between the particles.
The particles accelerate according to:

$$a = \frac{F}{m}.$$

The net force on particle *a* is the sum of all forces.



For computer simulation the forces are calculated at discrete times, $t_0, t_1, \ldots,$ with time intervals, δt. A particle that accelerates under a constant force over given time interval changes it velocity according to:

$$v' = v + \frac{F_{net}\delta t}{m}$$

where F_{net} is the net force on the particle. Since the force is not really constant over the time interval, this is an approximation. Similarly the position, x, of each particle is updated from the velocity:

$$x' = x + v\delta t.$$

Clearly we want δt to be small in order to avoid inaccuracies.

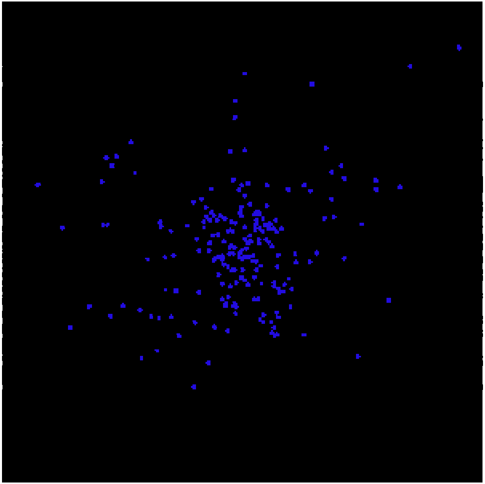
In three dimensional space the distance between two particles becomes

$$d = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}$$

and the force in the x-direction is

$$F_x = \frac{Gm_am_b}{r^2}\left(\frac{x_b - x_a}{r}\right)$$

with the other force components similarly calculated. The acceleration and positions are calculated independently in each direction.



200 particles experiencing gravitational force in three dimensional space.

```
for(t=0;t<tmax;t++){
  for(i=0;i<N;i++){
    F=Calc_force(i);
    vnew[i]=v[i]+F*dt/m;
    xnew[i]=x[i]+v[i]*dt;
  }
  for(i=0;i<N;i++){
    x[i]=xnew[i];
    v[i]=vnew[i];
  }
}
```

An intuitive parallelization has each of p processors responsible for N/p particles.

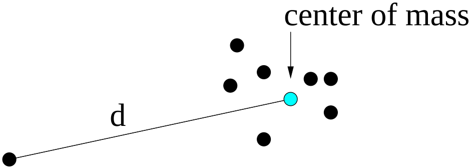
Each processor will then have to send N/p distinct units of data to each of the other $p-1$ processors after each round. This is effectively a gather/broadcast operation and suggests synchronous operation.

Fundamentally the message passing complexity is unavoidable without changing the computations.

Note also that the particles assigned to a processor will move and eventually could be considered in general to be uniformly distributed over the space.

Similarly, dividing space over the processors can lead to problems because the particles may all end up in a particular region of space and the computation degrades to a sequential one.

The complexity can be reduced by applying a mathematical approximation known as clustering. The clustering approximation tries to find particles that are relatively close together and treat them as a single particle.



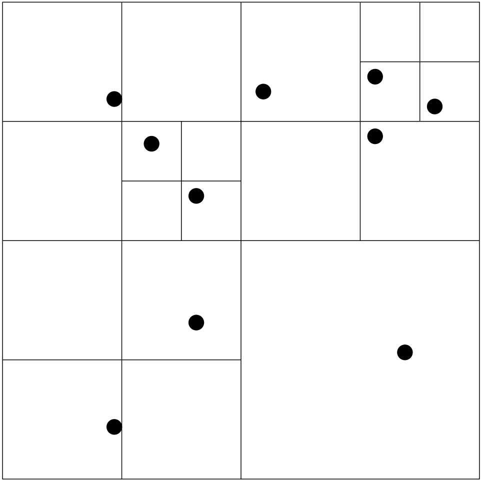
Barnes-Hut Algorithm

The Barnes-Hut algorithm uses divide-and-conquer to find clusters of particles in the N-body problem.

Consider the particles being contained within a 3 dimensional cube. An *octtree* is constructed by recursively subdividing the cube into 8 smaller cubes. If a cube does not contain a particle then it is discarded. A cube that contains exactly one particle becomes a leaf cube. All other cubes are further subdivided.

This subdivision process results in an incomplete tree, where each node in the tree has up to 8 children. Leaf nodes in the tree are cubes that contain exactly one particle.

The subdivision is easier to visualize in two dimensions, as a *quadtree*.



The Barnes-Hut algorithm calculates the total mass and the center of mass of each of the nodes in the tree.

The total mass of a node with only one particle is the mass of the particle.

The center of mass of a node with only one particle is the center of the particle.

For nodes with some number of children the total mass, M, is the sum of total mass, m_i, for each child i. The center of mass is

$$\frac{1}{M}\sum_i c_i m_i$$

where c_i is the center of mass for child i.

The force on each particle, i, is computed by traversing the tree from the root. If the center of mass of the root is further than

$$\frac{l}{\theta}$$

where l is the dimension of the cube and $0 < \theta \leq 1$ is an opening angle, from particle i then the root node is used to compute the force on particle i. If not then the algorithm is recursively applied to the children, summing up the forces obtained to obtain a net force.

The choice of opening angle determines a compromise between accuracy and computational requirements.

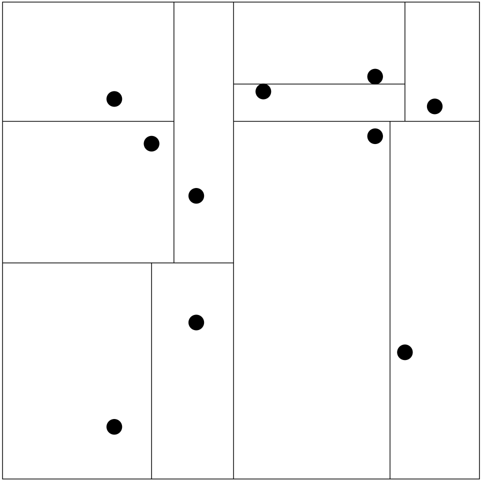
The approach is known to substantially reduce the computational requirements.

Orthogonal recursive bisection

The octtree or quadtree presented earlier does not lead to a balanced computation since some children will contain no particles.

An orthogonal recursive bisection attempts to provide a more balanced division of space.

In two dimensions the first division finds a vertical line that divides the particles into two equal groups. The second division finds a horizontal line that divides the two groups each into two equal groups and so on. The subdivision continues until either the number of groups equals the number of processors or there is one particle in each group.



Jacobi iteration

Given a general system of n linear equations,

$$\mathrm{Ax} = \mathrm{b},$$

the Jacobi iteration is described by:

$$x_i^k = \frac{1}{a_{i,i}}\left[b_i - \sum_{j\neq i} a_{i,j}x_j^{k-1}\right],$$

where the superscript k indicates the k-th iteration.
A direction solution requires $O(n^2)$ time with n processes. Solution via Jacobi iteration depends on the number of iterations and the required accuracy.

Laplace's equation

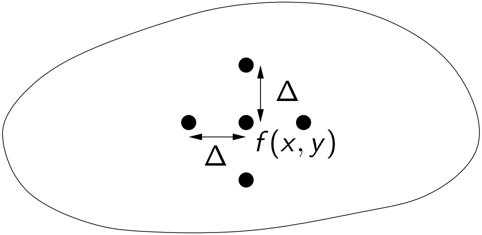
The Jacobi iteration technique can be used to solve Laplace's equation:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = 0$$

The equation needs to be solved for each point (x, y) within a given two dimensional region. Some points in the region are used to hold *boundary conditions*.

In this case, the solution space is discretized into a large number of points and the *finite difference* method is applied with Jacobi iteration.

Discretizing in space is similar to discretizing in time for the N-body problem.



$$\frac{\partial f}{\partial x} \approx \frac{1}{\Delta}\left[f(x+\frac{\Delta}{2},y) - f(x-\frac{\Delta}{2},y)\right]$$

$$\frac{\partial^2 f}{\partial x^2} \approx \frac{1}{\Delta}\Big[\frac{1}{\Delta}\big[f(x+\Delta,y) - f(x,y)\big] - \frac{1}{\Delta}\big[f(x,y) - f(x-\Delta,y)\big]\Big]$$

$$= \frac{1}{\Delta^2}\Big[f(x+\Delta,y) - 2f(x,y) + f(x-\Delta,y)\Big].$$

Similarly:

$$\frac{\partial^2 f}{\partial y^2} \approx \frac{1}{\Delta^2}\Big[f(x,y+\Delta) - 2f(x,y) + f(x,y-\Delta)\Big],$$

and substituting in Laplace's equation we get:

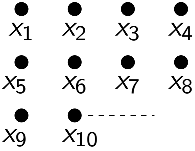
$$\frac{1}{\Delta^2}\Big[f(x+\Delta,y) + f(x-\Delta,y) + f(x,y+\Delta) + f(x,y-\Delta) - 4f(x,y)\Big] = 0.$$

Rearranging and rewriting using Jacobi iterative formula gives:

$$f^k(x,y) = \frac{1}{4}\Big[f^{k-1}(x-\Delta,y) + f^{k-1}(x,y-\Delta) + \\ f^{k-1}(x+\Delta,y) + f^{k-1}(x,y+\Delta)\Big].$$

Repeated application of the formula converges to the solution.

The points in an n^2 mesh can be labeled $x_1, x_2, \ldots, x_{n^2}$.



This natural ordering of the points leads to a number of linear equations with five unknowns:

$$x_{i-n} + x_{i-1} + 4x_i + x_{i+1} + x_{i+n} = 0.$$

The equation does not apply at the boundary points $(x_1, x_2, \ldots, x_4, x_5, x_8, x_9, \ldots)$.

$$
\begin{bmatrix}
\ddots & & & & & & & \\
& \ddots & & & & & & \\
& & \ddots & & & & & \\
& & & \ddots & & & & \\
& 1 & \cdots & 1 & -4 & 1 & \cdots & 1 \\
& & & & & \ddots & & \\
& & & & & & \ddots & \\
& & & & & & & \ddots
\end{bmatrix}
\times
\begin{bmatrix}
x_1 \\ x_2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ x_{n^2}
\end{bmatrix}
=
\begin{bmatrix}
0 \\ 0 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ 0
\end{bmatrix}
\tag{1}
$$

Figure: Sparse matrix for Laplace's equation.

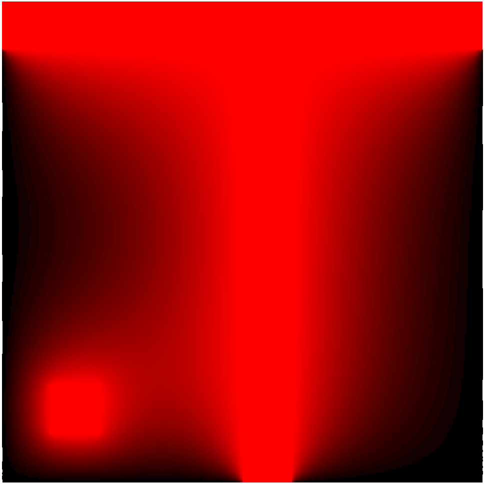


Figure: Heat gradient with a “T” strip and square, bounded by zero temperature.

The Jacobi iteration is useful for cases such as solving Laplace's equations because the matrix is sparse and so the iterative methods can use less memory and reach an acceptable solution quickly.

Jacobi iterations can converge in as little as $O(\log n)$ steps, where with N processes the parallel time complexity is $O(\log n)$, much better than using Gaussian elimination. However, Jacobi iterations can converge slowly for other types of matrices.

Gauss-Seidel relaxation

The Guass-Siedel relaxation attempts to increase the convergence rate by using values computed for the k-th iteration in subsequent computations within the k-th iteration. The general iteration formula is:

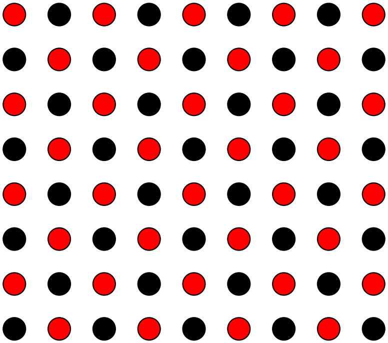
$$x_i^k = \frac{1}{a_{i,i}}\left[b_i - \sum_{j=1}^{i-1} a_{i,j}x_j^k - \sum_{j=i+1}^{n} a_{i,j}x_j^{k-1}\right],$$

which assume we compute the x_i in natural order (i.e. sequentially). Laplace's equation is solved using:

$$\begin{aligned} f^k(x,y) = \tfrac{1}{4}\Big[&f^k(x-\Delta,y) + f^k(x,y-\Delta) + \\ &f^{k-1}(x+\Delta,y) + f^{k-1}(x,y+\Delta)\Big]. \end{aligned}$$

Red-Black ordering

All red points are calculated in parallel using the black points. Then all the black points are calculated in parallel using the red points.



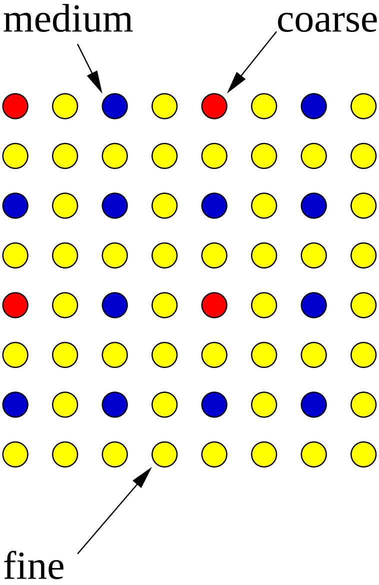
A variation of red-black ordering is *red-black checkerboard*. In this case the space is divided into square regions that are painted red and black and the computation proceeds similarly.

Multi-grid technique

The multi-grid technique computes the solution on a coarse grid first, then progressively on finer grids.

When the next finer grid is used, the initial points are computed by interpolation.

It is suggested that a suitable number of levels is $\log_2 n$, such that grids of size $2^t \times 2^t$ are used for $t = 1, 2, \ldots, \log_2 n$.


medium
coarse
fine

Centralised Program Termination

Stopping the computation when the solution has been reached is called *termination*. For a distributed program with a centralised style of load balancing, determining program termination is quite striaghtforward: the master process recognises and decides when to terminate.
If a slave process finds the solution, it can send a message to the master which in turn stops all others.
Similarly, for a computation in which tasks are taken from a task queue, the computation terminates when both of the following are satisfied:

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

However, when task distribution is decentralised, determining when the program should terminate becomes a non-trivial task.

Decentralised Program Termination

If a single process always finds a "solution", termination conditions are simple. However, not all computations end with a particular solution being found. In the general case, distributed termination at time t requires the following conditions to be satisfied:

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

Note in particular the requirement that no messages are in transit (as they may restart processes that have terminated locally).

One could conceivably wait long enough after local termination to allow any message in transit to arrive. However, this approach is both ineffecient and would not be portable between different machines and architectures.

Acknowledgement Messages

Use request/acknowledge messages. Each process has two states, *active* and *inactive*. Each processor, P_i, uses the following logic:

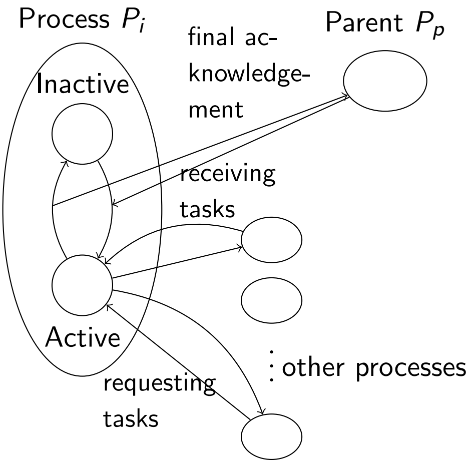
1. Initiate as inactive, wait to receive task from another process.
2. When receving its *first* task, from another process P_p, switch to active state. P_p is set to parent of P_i.
3. While active, P_i may send tasks to other processes and always expects an acknowledgement message.
4. If receiving a task from another process, immediately send an acknowledgement message (except to P_p).
5. When ready to become inactive, send acknowledgment message to parent.

A process is ready to become inactive when:

- Local termination conditions are met (all tasks completed)
- Transmitted acknowledgements for every task received
- Received acknowledgements for every task sent out

Things to note, following this acknowledgement message model:

- The last condition means that a process must become inactive before its parent process
- The “termination structure” forms a tree, but the computation need not be one, an active process can send tasks to any other process, becoming the parent of previously inactive processes


Process P_i
final acknowledgement
Parent P_p
Inactive
receiving tasks
Active
other processes
requesting tasks

Ring Termination Algorithm

Another approach is to organise the processes into a ring structure (particularly if the underlying architecture is a ring). There are two situations to consider:

- Processes not allowed to reactivate after reaching local termination - single pass ring termination
- Processes allowed to reactive - dual pass ring termination

In both cases, label p processes $P_0, P_1, \ldots, P_{p-1}$ where P_0 becomes the central point for global termination. Also, all message passes expect acknowledgement messages.

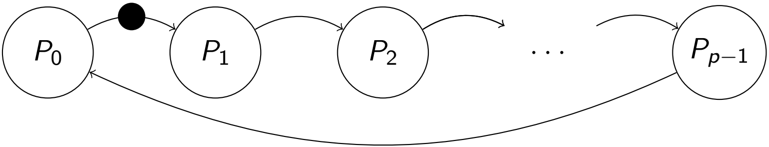
For single pass ring termination:

1. When P_0 has terminated it generates a token that is passed to P_i
2. When $P_i(1 \leq i < p)$ receives the token *and* has already termintaed, it passes the token onward to P_{i+1}. Otherwise, it waits for its local termination condition then passes the token. P_{p-1} passes the token to P_0.
3. When P_0 receives the token, it knows that all processes have terminated and can pass a message to all other processes to inform them of global termination if necessary.

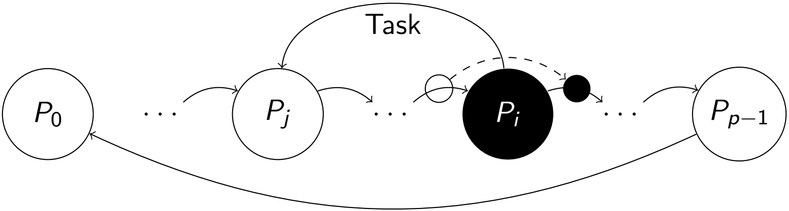
For dual pass ring termination, each process and token is coloured white or black:

1. P_0 becomes white when it has terminated and generates a white token to P_1
2. As before, the token is passed through the ring from one process to the next when each process has terminated, but the colour of the token may be changed. If P_i sends a *task* (not token) to P_j where $j < i$ (earlier in the ring), P_i becomes a black process; otherwise it is white. A black process will colour a token black and pass it on. A white process will pass on the token in its original colour. After P_i has passed on a token, it becomes white.
3. When P_0 receives a black token, it passes on a white token; if it receives a white token, all processes have terminated.

By this algorithm, a black process represents a process unsure if some process before it has terminated, colouring the token black to inform P_0 that another pass is necessary. It then turns white as it will only receive a token after the task it sent to P_j was completed. If every process is white, the original white token sent out by P_0 is returned white and all process have terminated.



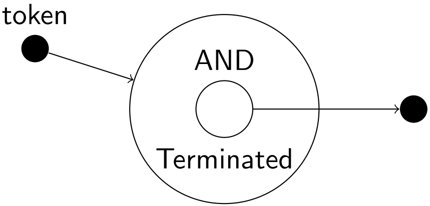
Single pass ring termination



Dual pass ring termination

Tree Termination Algorithm

The local action of each processor as described for the ring can be applied to other interconnection structures, notably a tree. Following this structure, any particular process can know when all its descendants have terminated locally and when the root receives tokens from all its children, the global termination process can begin.



Unit process algorithm for local termination

Fixed Energy Distribution Termination

Algorithm using the notion of a fixed quantity within the system. Similar to tokens with numeric value.

- Initially, all energy held by a single, master process.
- Each process can hand out any energy it has to other process when requesting a task be done, representing the workload being shared.
- Idle processes pass energy back either directly to the master or to the process that requested the work to be done.
- One significant disadvantage is finite precision and adding partial energies may not equate to the original energy
- One possible solution is to use a large enough integer to cope with the number of divisions

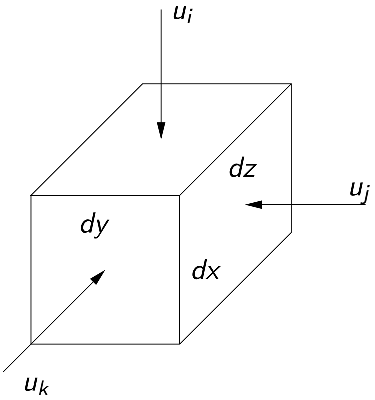
Navier-Stokes equation

The Navier-Stokes equation describes incompressible fluid flow, where the flow is relatively slow.

As with Laplace's equation, the fluid flow computation uses finite element analysis techniques and discretizes space.

Each unit of space records some description of the fluid in that unit.

Typically the cell will record the velocities of the fluid, at all six faces of the unit cube, and the pressure of the fluid at the center of the unit cube.



The simulation involves (among other things):

- maintaining boundary conditions,
- computing solution to Navier-Stokes equation,
- tracking the surface of the fluid and
- solving the continuity equation.

The overall computational steps are too length to include in this subject.
You may be interested to parallelize the fluid flow computation as a summer project.
Fluid flow computations provide excellent potential for parallelization.