UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

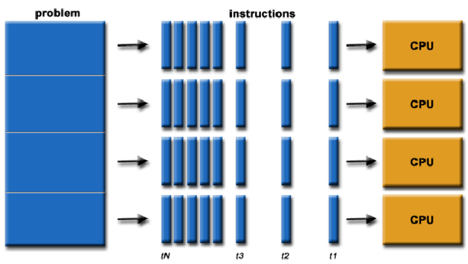
Lecture 4:
Embarrassingly Parallel Computations

Parallel Programming (INF-3201)

John Markus Bjørndalen

Parallel computations

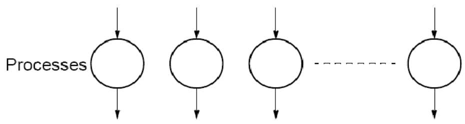
- A program is run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs



Source: Blaise Barney, "Introduction to Parallel Computing", Livermore Computing.

Embarrassingly Parallel Computations

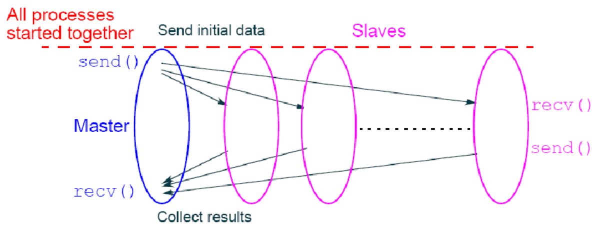
A computation that can obviously be divided into a number of completely independent parts, each of which can be executed by a separate process(or).



No communication or very little communication between processes

- Each process can do its tasks without any interaction with other processes
- Speedup, message-passing, SPMD

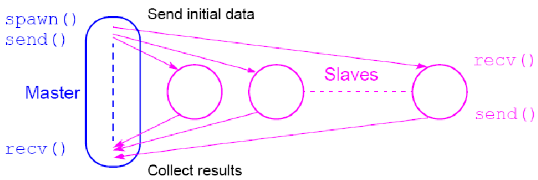
Static process creation Master-slave approach



Usual MPI approach

Dynamic process creation Master-slave approach

Start Master initially



(PVM approach)

Example: Low level image processing

- Operates directly on stored image to improve/enhance it
- Stored image consists of two-dimensional array of *pixels* (picture elements/"image dots")

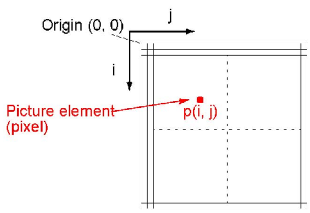
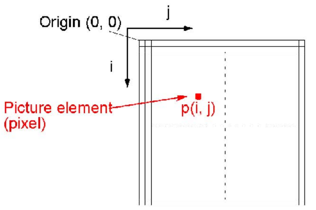


Figure from Barry Wilkinson & Michael Allen. Parallel Programming

Example: Low level image processing

Many operations are embarassingly parallel

Example: operations that compute each pixel independently
(such as shift, scale, rotate)



Some geometrical operations

Shifting

Object shifted by Δx in the x-dimension and Δy in the y-dimension:

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

where x and y are the original and x' and y' are the new coordinates.

Scaling

Object scaled by a factor S_x in x-direction and S_y in y-direction:

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

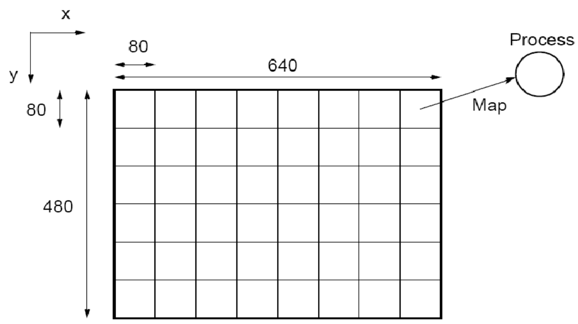
Some geometrical operations

Rotation

Object rotated through an angle θ about the origin of the coordinate system:

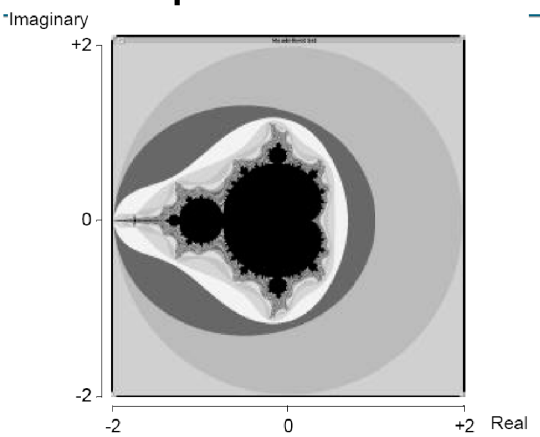
$$x' = x\cos\theta + y\sin\theta$$
$$y' = -x\sin\theta + y\cos\theta$$

Partitioning into regions for individual processes



Square region for each process (can also use strips)

Example: Mandelbrot set



Mandelbrot Set

Set of points c in a complex plane that are quasi-stable (will increase and decrease, but not exceed some limit) when computed by iterating the function

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

where z_{k+1} is the $(k+1)th$ iteration of the complex number $z = a + bi$. The initial value for z is zero.

c is a complex number giving position of point in the complex plane.

Iterations continued until magnitude of z is greater than 2 or number of iterations reaches arbitrary limit.

Magnitude of z is the length of the vector given by

$$z_{length} = \sqrt{a^2 + b^2}$$

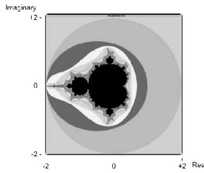
Sequential routine computing value of one point returning number of iterations

```c
struct complex {
   float real;
   float imag;
};
int cal_pixel(complex c)
{
   int count, max;
   complex z;
   float temp, lengthsq;
   max_iter = 256;                    /* 256 colors */
   z.real = 0; z.imag = 0;
   count = 0;                         /* number of iterations */
   do {
      temp = z.real * z.real - z.imag * z.imag + c.real;
      z.imag = 2 * z.real * z.imag + c.imag;
      z.real = temp;
      lengthsq = z.real * z.real + z.imag * z.imag;
      count++;
   } while ((lengthsq < 4.0) && (count < max_iter));
   return count;               /* color of the pixel */
}
```


Parallelizing Mandelbrot Set Computation

Static Task Assignment

Simply divide the region in to fixed number of parts, each computed by a separate processor.

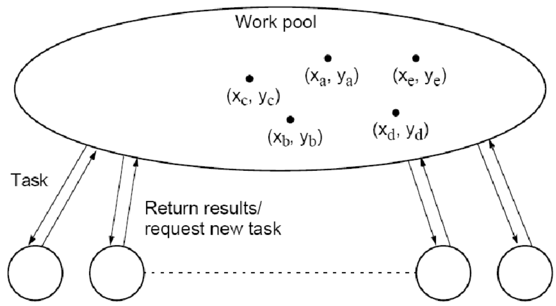


Not very successful because different regions require different numbers of iterations and time.

Dynamic Task Assignment

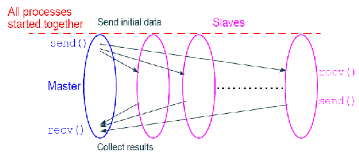
Have processor request regions after computing previous regions

Dynamic Task Assignment
Work Pool/Processor Farms



Parallel computation

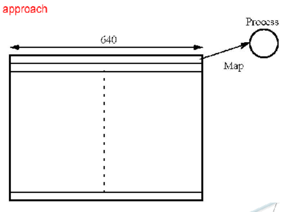
Assumption
- # processors/processes is given (num_proc)



- Each processor computes 1 row at a time
  - Communication time

    $t_{comm} = t_{startup} + nt_{data}$
  - The work pool holds row numbers

Usual MPI approach



Master code

```c
count = 0;                                  /* counter for termination*/
row = 0;                                    /* row being sent */
for (k = 0; k < num_proc; k++) {            /* assuming num_proc < disp_height */
    send(row, Pk, data_tag);                /* send initial row to process */
    count++;                                /* count rows sent */
    row++;
} /* next row */
do {
    recv (&slave, &r, &color, PANY, result_tag);
    count--;                                /* reduce count as rows received */
    if (row < disp_height) {
        send (row, Pslave, data_tag);       /* send next row */
        row++;                              /* next row */
        count++;
    } else
        send (row, Pslave, terminator_tag); /* terminate */
    display (r, color);                     /* display row */

} while (count > 0);
```


Slave code

```c
recv(&y, Pmaster, ANYTAG, source_tag);           /* receive 1st row to compute */
while (source_tag == data_tag) {
    c.imag = y;
    for (x = 0; x < disp_width; x++) {            /* compute row colors */
        c.real = x;
        color[x] = cal_pixel(c);
    }
    send(pid, y, color, Pmaster, result_tag);    /* row colors to master */
    recv(&y, Pmaster, source_tag);               /* receive next row */
};
```

Monte Carlo Methods

Another embarrassingly parallel computation.

Monte Carlo methods use of random selections.

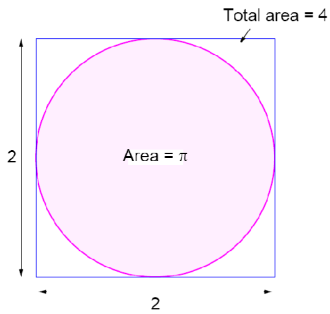
Example - To calculate π

Circle formed within a 2 x 2 square. Ratio of area of circle to square given by:

$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi(1)^2}{2 \times 2} = \frac{\pi}{4}$$

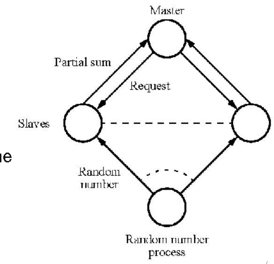
Points within square chosen randomly. Score kept of how many points happen to lie within circle.

Fraction of points within the circle will be $\frac{\pi}{4}$, given a sufficient number of randomly selected samples.



Parallel implementation

- Observation
  - Independent iterations ⇒ embarrassingly parallel problem
- Concern
  - Each computation must use a different random number, and
  - No correlation between the random numbers.
- Approach
  - Have a process responsible for issuing the next random number
  - (not a good one in practice – why?)



Parallel implementation

Master

```
for(i = 0; i < N/n; i++) {
   for (j = 0; j < n; j++)              // n= #random numbers for slave
       xr[j] = rand();                   // load numbers to be sent
   recv(P_ANY, req_tag, P_source);       // wait for a slave to make request
   send(xr, n, P_source, compute_tag);
}
for(i = 0; i < num_slaves; i++) {       // terminate computation
   recv(P_i, req_tag);
   send(P_i, stop_tag);
}
sum = 0;
reduce_add(&sum, P_group);              // collective routine
```

Slaves

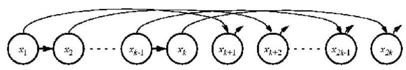
```
sum = 0;
send(P_master, req_tag);
recv(&xr, &n, P_master, source_tag);
while (source_tag == compute_tag) {
    for (i = 0; i < n; i++)
        sum = sum + xr[i] * xr[i] - 3 * xr[i];
    send(P_master, req_tag);
    recv(&xr, &n, P_master, source_tag);
};
reduce_add(&sum, P_group);              //Collective routine
```


Parallel implementation

- May not scale well
  - Random numbers computed one location
  - Random numbers sent to each location
- Still shows principle of handing out tasks and terminating

Random number generation

- Goal
  - Create pseudorandom-number sequence $x_1, x_2, \ldots, x_i, x_{i+1}, \ldots, x_n$
- Sequential version
  - Evaluate x_{i+1} from a function f of x_i
  - f must create a large sequence with correct statistical properties
  - Regular form: $x_{i+1} = (ax_i + c) \bmod m$
- Parallel version
  - Observation: $x_{i+k} = (Ax_i + C) \bmod m$
    - $A = a^k \bmod m$; $C = c(a^{k-1} + a^{k-2} + \ldots + a^1 + a^0) \bmod m$ (cf. next slide)
    - k: a selected jump constant (usually, #processors)



Available library: Scalable Parallel Random Number Generators (SPRNG)