Parallel Techniques

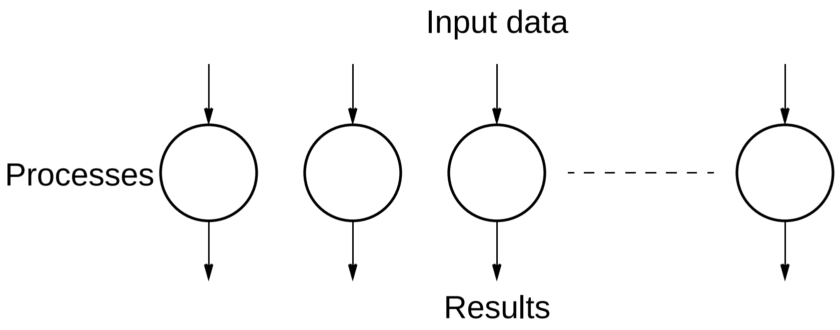
- Embarrassingly Parallel Computations
- Partitioning and Divide-and-Conquer Strategies
- Pipelined Computations
- Synchronous Computations
- Asynchronous Computations
- Load Balancing and Termination Detection

Chapter 3

Embarrassingly Parallel Computations

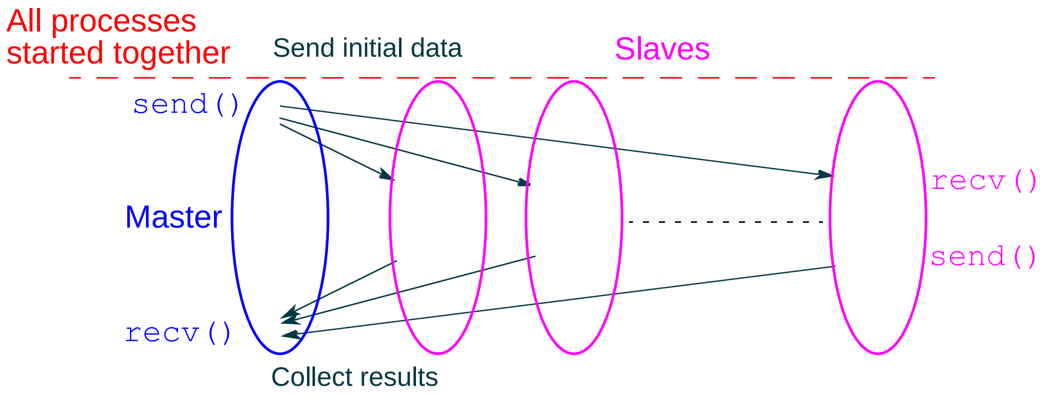
Embarrassingly Parallel Computations

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

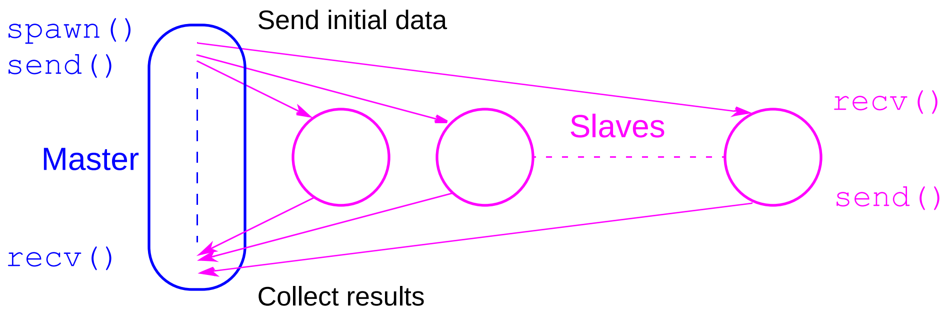
Practical embarrassingly parallel computation with static process creation and master-slave approach



Usual MPI approach

Practical embarrassingly parallel computation with dynamic process creation and master-slave approach

Start Master initially



(PVM approach)

Embarrassingly Parallel Computation Examples

- Low level image processing
- Mandelbrot set
- Monte Carlo Calculations

Low level image processing

Many low level image processing operations only involve local data with very limited if any communication between areas of interest.

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:

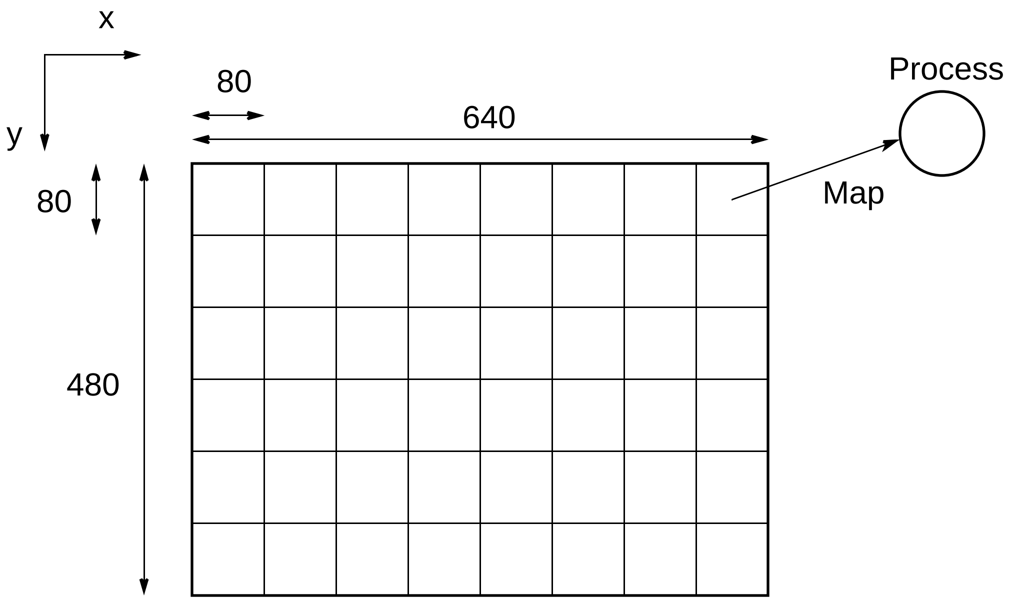
$$x' = xS_x$$
$$y' = yS_y$$

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)

Master

```
for (i = 0, row = 0; i < 48; i++, row = row + 10)  /* for each process*/
   send(row, Pi);                                  /* send row no.*/

for (i = 0; i < 480; i++)                          /* initialize temp */
   for (j = 0; j < 640; j++)
      temp_map[i][j] = 0;

for (i = 0; i < (640 * 480); i++) {                /* for each pixel */
   recv(oldrow,oldcol,newrow,newcol, PANY);        /* accept new coords */
   if !((newrow < 0)||(newrow >= 480)||(newcol < 0)||(newcol >= 640))
      temp_map[newrow][newcol]=map[oldrow][oldcol];
}
for (i = 0; i < 480; i++)                          /* update bitmap */
   for (j = 0; j < 640; j++)
      map[i][j] = temp_map[i][j];
```

Slave

```
recv(row, P_master);                                      /* receive row no. */
for (oldrow = row; oldrow < (row + 10); oldrow++)
   for (oldcol = 0; oldcol < 640; oldcol++) {             /* transform coords */
      newrow = oldrow + delta_x;                          /* shift in x direction */
      newcol = oldcol + delta_y;                          /* shift in y direction */
      send(oldrow,oldcol,newrow,newcol, P_master);          /* coords to master */
   }
```

Mandelbrot Set

Set of points 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 $z = a + bi$ and c is a complex number giving position of point in the complex plane. The initial value for z is zero.

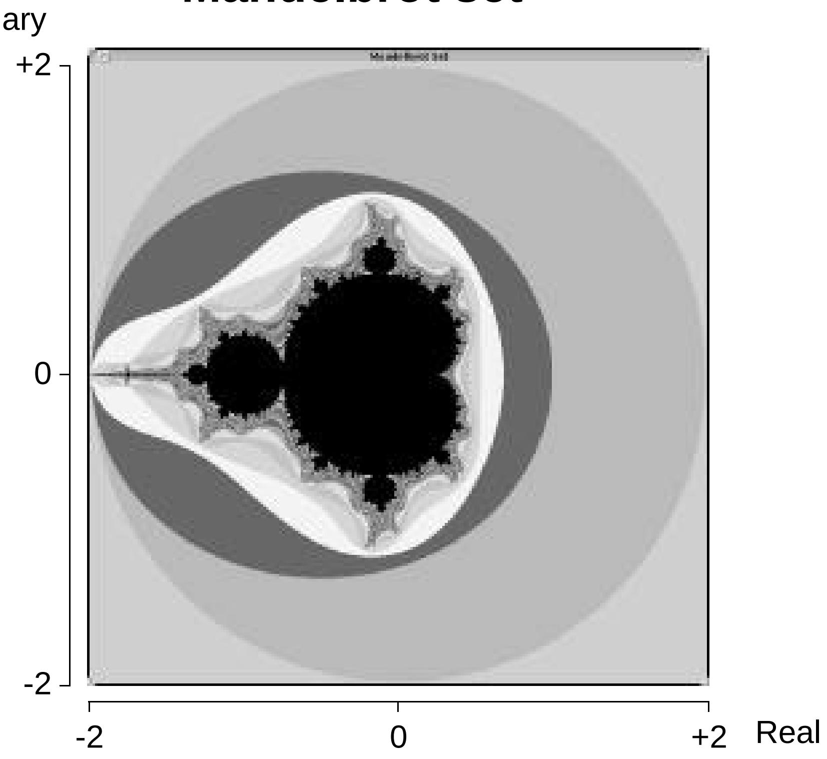
Iterations continued until magnitude of z is greater than 2 or number of iterations reaches arbitrary limit. Magnitude of z is the length of the vector given by

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

Sequential routine computing value of one point returning number of iterations

```
structure complex {
  float real;
  float imag;
};
int cal_pixel(complex c)
{
int count, max;
complex z;
float temp, lengthsq;
max = 256;
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));
return count;
}
```

Mandelbrot set



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

Master

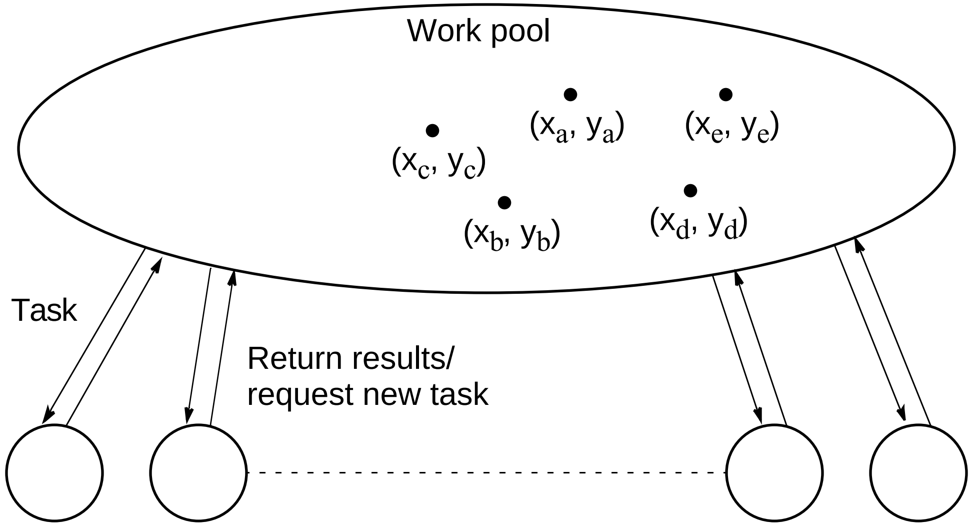
```
for (i = 0, row = 0; i < 48; i++, row = row + 10)/* for each process*/
   send(&row, P_i);                                   /* send row no.*/
for (i = 0; i < (480 * 640); i++) {   /* from processes, any order */
   recv(&c, &color, P_ANY);            /* receive coordinates/colors */
   display(c, color);                  /* display pixel on screen */
}
```

Slave (process i)

```
recv(&row, P_master);                  /* receive row no. */
for (x = 0; x < disp_width; x++)       /* screen coordinates x and y */
   for (y = row; y < (row + 10); y++) {
      c.real = min_real + ((float) x * scale_real);
      c.imag = min_imag + ((float) y * scale_imag);
      color = cal_pixel(c);
      send(&c, &color, P_master);      /* send coords, color to master */
   }
```

Dynamic Task Assignment

Work Pool/Processor Farms



Master

```
count = 0;                                    /* counter for termination*/
row = 0;                                      /* row being sent */
for (k = 0; k < num_proc; k++) {            /* assuming num_proc<disp_height */
    send(&row, P_k, data_tag);                /* send initial row to process */
    count++;                                  /* count rows sent */
    row++;                                    /* next row */
}

do {
    recv (&slave, &r, color, P_ANY, result_tag);
    count--;                                /* reduce count as rows received */
    if (row < disp_height) {
        send (&row, P_slave, data_tag);                /* send next row */
        row++;                                         /* next row */
        count++;
    } else
        send (&row, P_slave, terminator_tag);          /* terminate */
    display (r, color);                                /* display row */
} while (count > 0);
```

Slave

```
recv(y, P_master, ANYTAG, source_tag);     /* receive 1st row to compute */
while (source_tag == data_tag) {
   c.imag = imag_min + ((float) y * scale_imag);
   for (x = 0; x < disp_width; x++) {            /* compute row colors */
      c.real = real_min + ((float) x * scale_real);
      color[x] = cal_pixel(c);
   }

  send(&x, &y, color, P_master, result_tag);      /* row colors to master */
  recv(&y, P_master, source_tag);                 /* receive next row */
};
```

Monte Carlo Methods

Another embarrassingly parallel computation.

Monte Carlo methods use of random selections.

Example - To calculate π

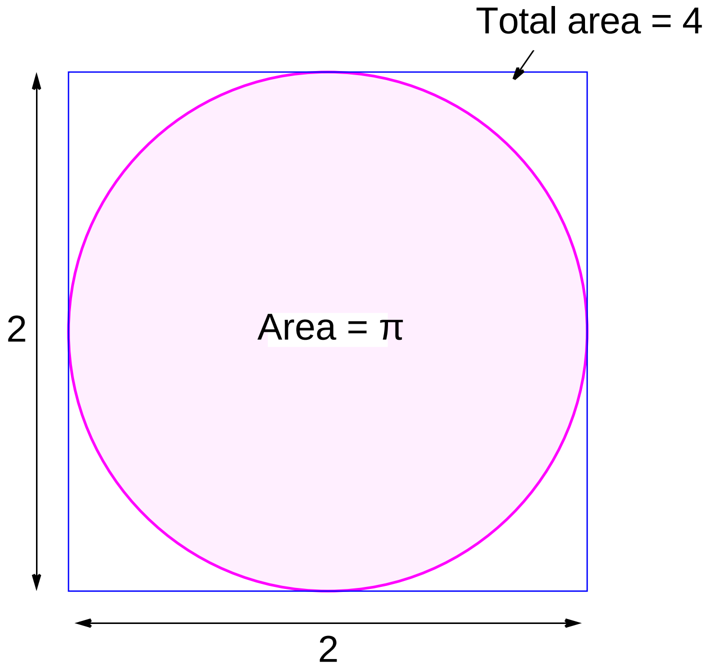
Circle formed within a square, with unit radius so that square has sides 2 × 2. Ratio of the area of the circle to the square given by

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

Points within square chosen randomly.

Score kept of how many points happen to lie within circle.

Fraction of points within the circle will be π/4, given a sufficient number of randomly selected samples.


Total area = 4
Area = π
2
2

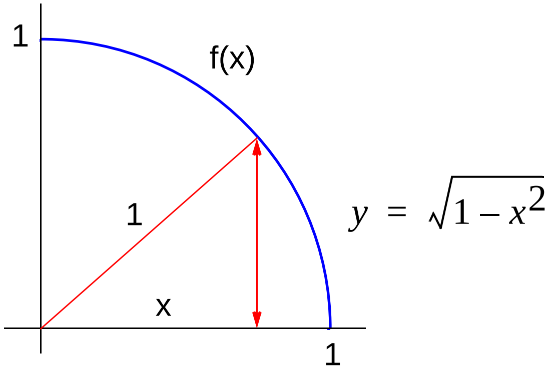
Computing an Integral

One quadrant of the construction can be described by integral

$$\int_0^1 \sqrt{1 - x^2}\, dx = \frac{\pi}{4}$$

Random pairs of numbers, (x_r, y_r) generated, each between 0 and 1.

Counted as in circle if $y_r \le \sqrt{1 - x_r^2}$; that is, $y_r^2 + x_r^2 \le 1$.



Alternative (better) Method

Use random values of *x* to compute *f*(*x*) and sum values of *f*(*x*):

$$\text{Area} = \int_{x_1}^{x_2} f(x)\,dx = \lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} f(x_r)(\mathrm{x}_2 - \mathrm{x}_1)$$

where x_r are randomly generated values of *x* between x_1 and x_2.

Monte Carlo method very useful if the function cannot be integrated numerically (maybe having a large number of variables)

Example

Computing the integral

$$I = \int_{x_1}^{x_2} (x^2 - 3x)\,dx$$

Sequential Code

```
sum = 0;
for (i = 0; i < N; i++) {  /* N random samples */
    xr = rand_v(x1, x2);    /* generate next random value */
    sum = sum + xr * xr - 3 * xr;    /* compute f(xr) */
}
area = (sum / N) * (x2 - x1);
```

Routine randv(x1, x2) returns a pseudorandom number between x1 and x2.

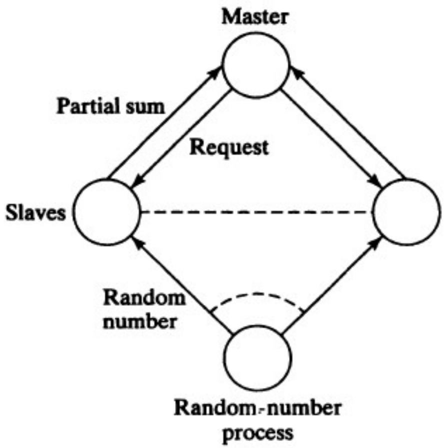


Figure 3.9 Parallel Monte Carlo integration.

Master

```
for(i = 0; i < N/n; i++) {
   for (j = 0; j < n; j++)        /*n=number of 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);
```

Slave

```
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, $P_{group}$);
```