CSC4005 – Distributed and Parallel Computing

Prof. Yeh-Ching Chung

School of Data Science

Chinese University of Hong Kong, Shenzhen

Outline

- Introduction to Parallel Computers
- Message Passing Computing and Programming
- Multithreaded Programming
- CUDA Programming
- OpenMP Programming
- Embarrassingly Parallel Computations
- Partitioning and Divide-and-Conquer Strategies
- Pipelined Computations
- Synchronous Computations
- Load Balancing and Termination Detection
- Sorting Algorithms

Embarrassingly Parallel Computations (1)

A computation that can be divided into a number of completely independent parts, each of which can be executed by a separate processor.

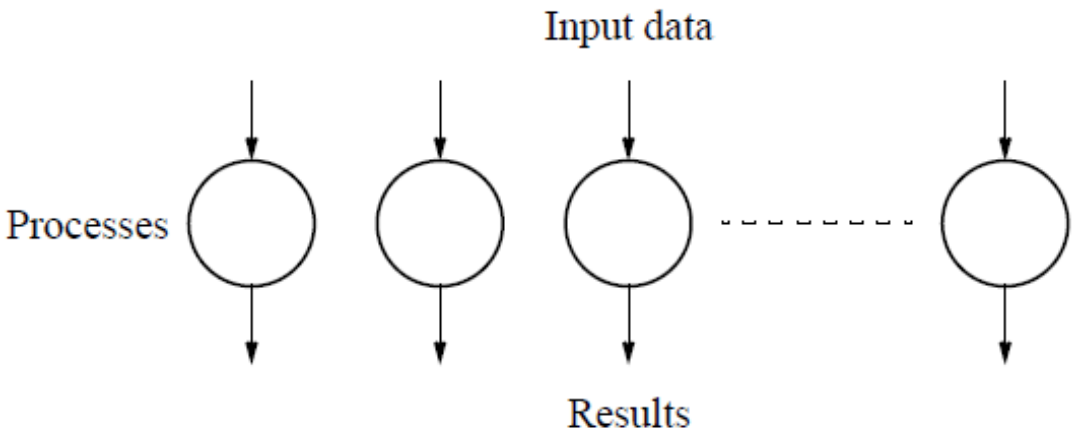


Figure 3.1 Disconnected computational graph (embarrassingly parallel problem).

Embarrassingly Parallel Computations (2)

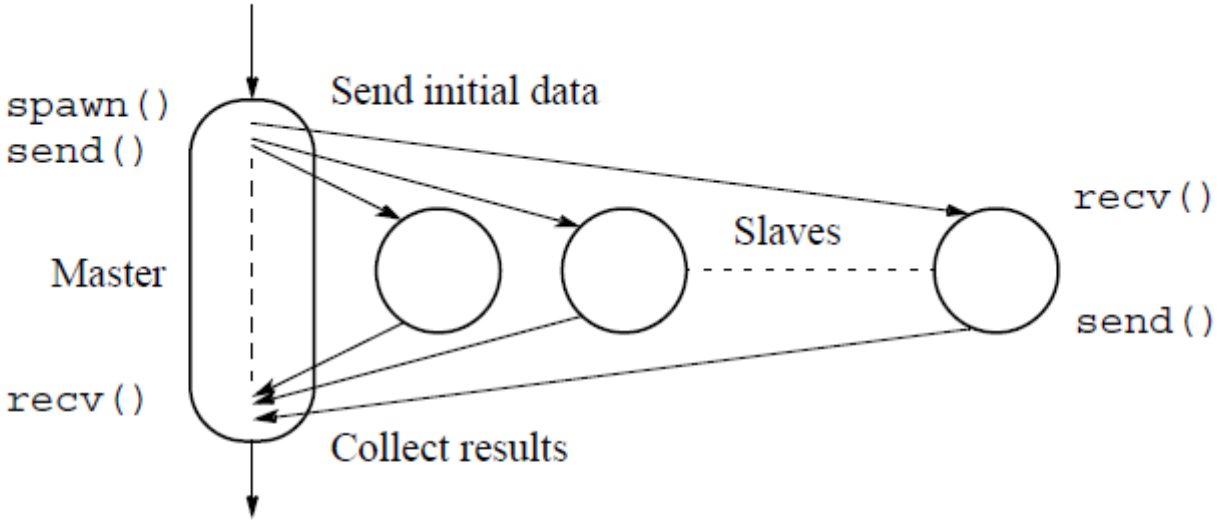


Figure 3.2 Practical embarrassingly parallel computational graph with dynamic process creation and the master-slave approach.

Embarrassingly Parallel Examples (1)

Low level image operations:

(a) 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.

(b) Scaling

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

$$x' = xS_x$$

$$y' = yS_y$$

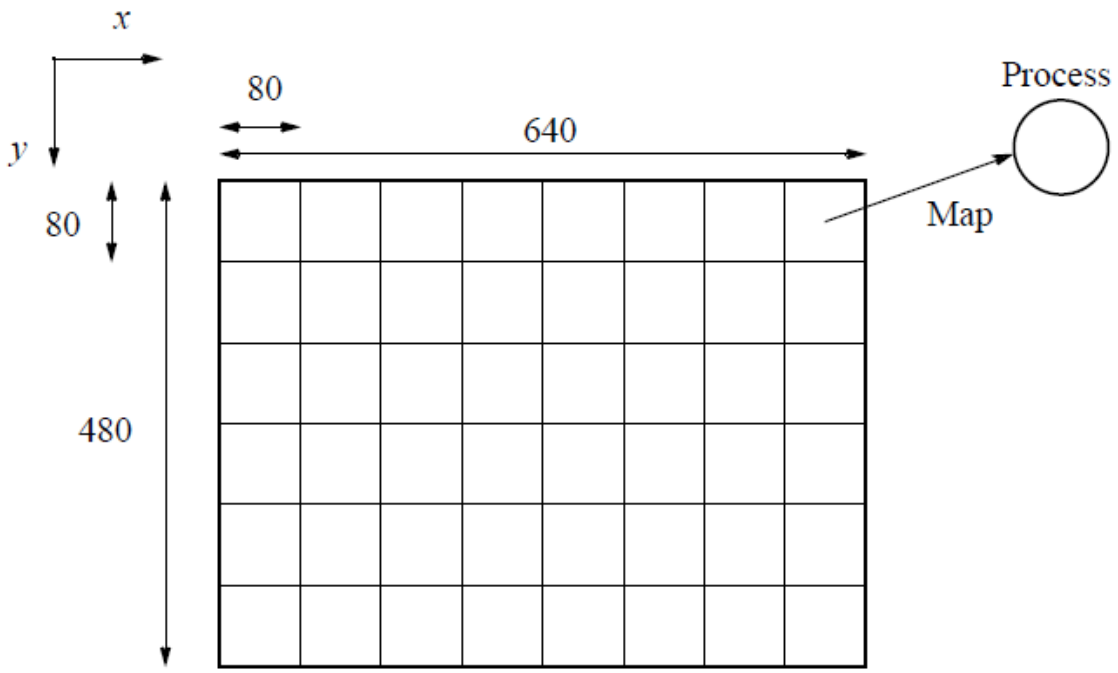
(c) Rotation

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

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

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

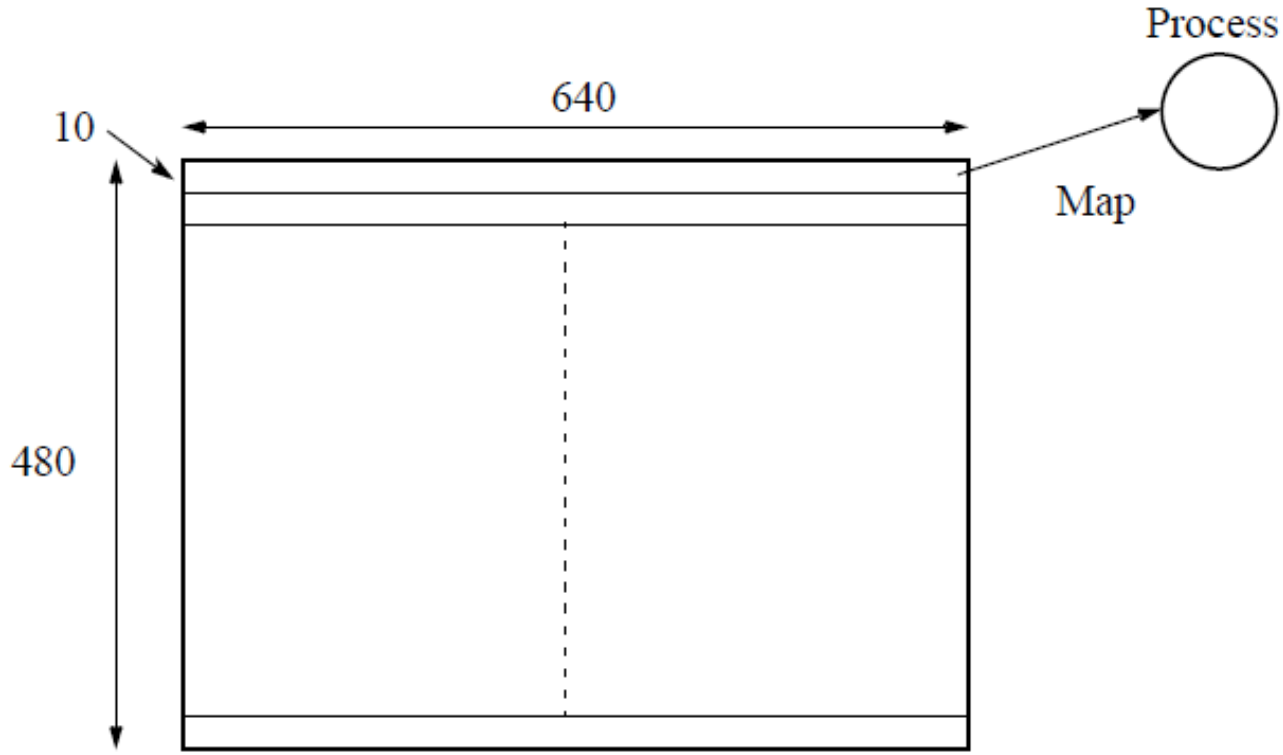
Embarrassingly Parallel Examples (2)



(a) Square region for each process

Figure 3.3 Partitioning into regions for individual processes.

Embarrassingly Parallel Examples (3)



(b) Row region for each process

Pseudocode to Perform Image Shift (1)

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];
```

Embarrassingly Parallel Examples (2)

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 */
    }
```

Embarrassingly Parallel Examples (3)

Analysis

Sequential

$$t_s = n^2 = O(n^2)$$

Parallel

Communication

$$t_{comm} = t_{startup} + mt_{data}$$

$$t_{comm} = p(t_{startup} + 2t_{data}) + 4n^2(t_{startup} + t_{data}) = O(p + n^2)$$

Computation

$$t_{comp} = 2\left(\frac{n^2}{p}\right) = O(n^2/p)$$

Overall Execution Time

$$t_p = t_{comp} + t_{comm}$$

For constant p, this is $O(n^2)$. However, the constant hidden in the communication part far exceeds those constants in the computation in most practical situations.

Mandelbrot Set Computation (1)

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 the position of the 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

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

Mandelbrot Set Computation (2)

Computing the complex function, $z_{k+1} = z_k^2 + c$, is simplified by recognizing that

$$z^2 = a^2 + 2abi + bi^2 = a^2 - b^2 + 2abi$$

or a real part that is $a^2 - b^2$ and an imaginary part that is $2ab$.

The next iteration values can be produced by computing:

$$z_{\text{real}} = z_{\text{real}}^2 - z_{\text{imag}}^2 + c_{\text{real}}$$

$$z_{\text{imag}} = 2z_{\text{real}}z_{\text{imag}} + c_{\text{imag}}$$

Mandelbrot Set Computation (3)

Seq. Routine computing value of one pt, returning no 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 Computation (4)

Scaling Coordinate System

For computational efficiency, let

```
scale_real = (real_max - real_min)/disp_width;
scale_imag = (imag_max - imag_min)/disp_height;
```

Including scaling, the code could be of the form

```
for (x = 0; x < disp_width; x++) /* screen coordinates x and y */
  for (y = 0; y < disp_height; y++) {
    c.real = real_min + ((float) x * scale_real);
    c.imag = imag_min + ((float) y * scale_imag);
    color = cal_pixel(c);
    display(x, y, color);
  }
```

where `display()` is a routine to display the pixel (x, y) at the computed color.

Mandelbrot Set Computation (5)

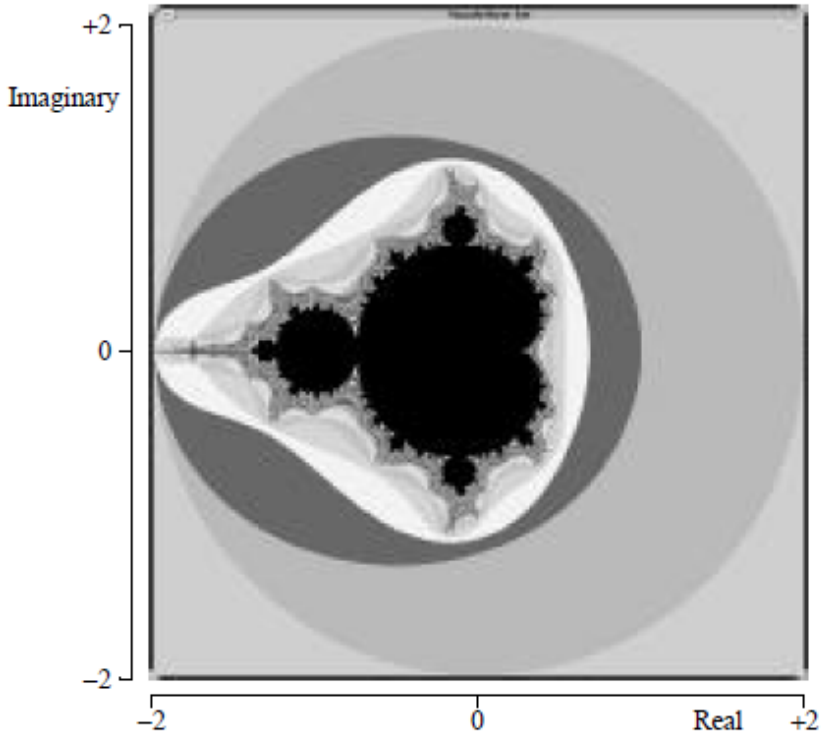


Figure 3.4 Mandelbrot set.

Parallelization of Mandelbrot Computation (1)

Static Task Assignment

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 */
    }
```

Parallelization of Mandelbrot Computation (2)

Dynamic Task Assignment

Work Pool/Processor Farms

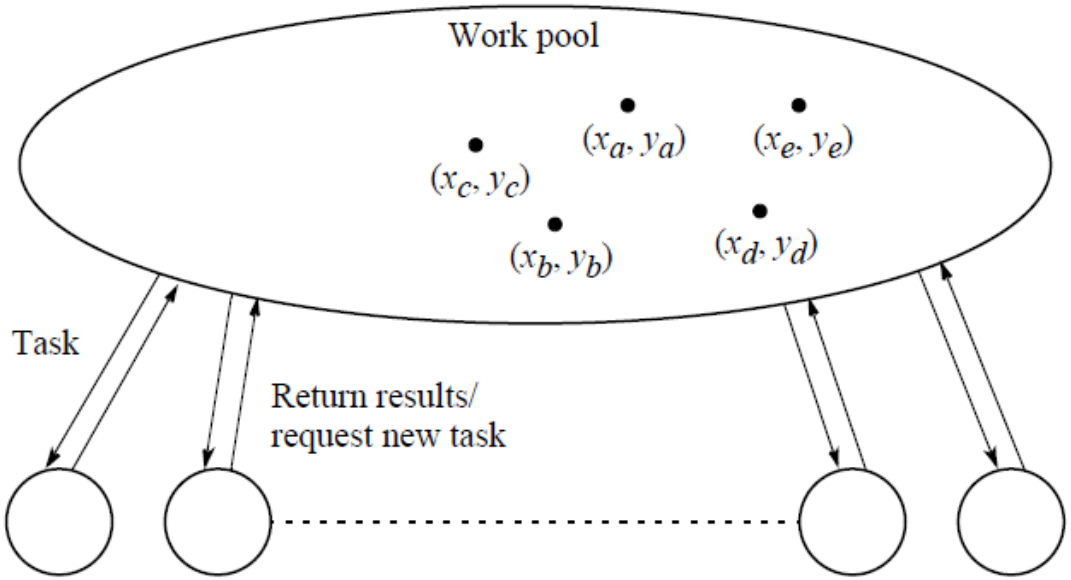


Figure 3.5 Work pool approach.

Parallelization of Mandelbrot Computation (3)

Coding for Work Pool Approach

Master

```
count = 0;                          /* counter for termination*/
row = 0;                            /* row being sent */
for (k = 0; k < procno; k++) {      /* assuming procno<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 */
    rows_recv++;
    display (r, color);                     /* display row */
} while (count > 0);
```

Parallelization of Mandelbrot Computation (4)

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

Parallelization of Mandelbrot Computation (5)

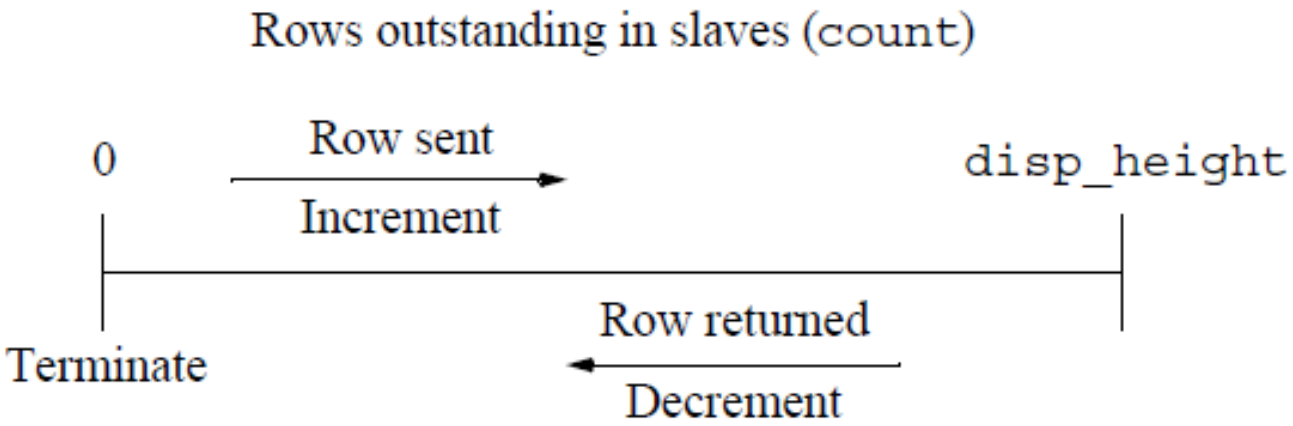


Figure 3.6 Counter termination.

Parallelization of Mandelbrot Computation (6)

Analysis

Sequential

$$t_s \leq \max \times n = O(n)$$

Parallel program

Phase 1: Communication - Row number is sent to each slave

$$t_{comm1} = s(t_{startup} + t_{data})$$

Phase 2: Computation - Slaves perform their Mandelbrot computation in parallel

$$t_{comp} \leq \frac{\max \times n}{s}$$

Phase 3: Communication - Results passed back to master using individual sends

$$t_{comm2} = \frac{n}{s}(t_{startup} + t_{data})$$

Overall

$$t_p \leq \frac{\max \times n}{s} + \left(\frac{n}{s} + s\right)(t_{startup} + t_{data})$$

Monte Carlo Methods (1)

Basis of Monte Carlo methods is the use of random selections in calculations.

Example - To calculate π

A circle is formed within a square. Circle has unit radius so that square has sides 2×2.

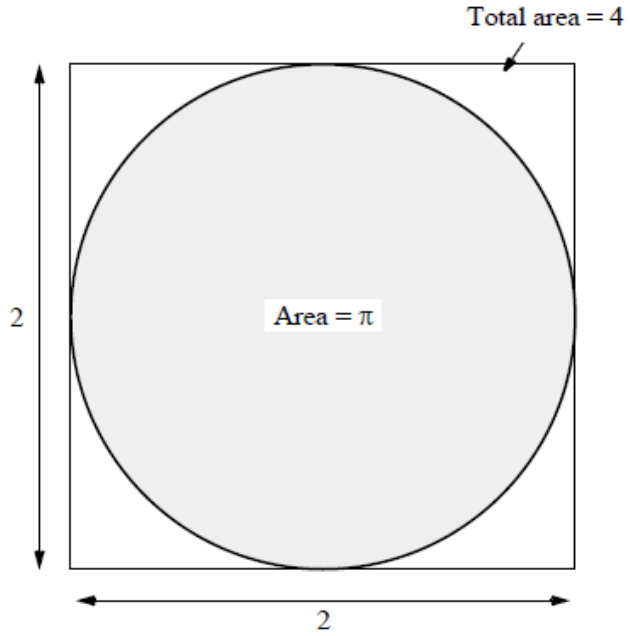


Figure 3.7 Computing π by a Monte Carlo method.

Monte Carlo Methods (2)

The ratio of the area of the circle to the square is given by

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

Points within the square are chosen randomly and a score is kept of how many points happen to lie within the circle.

The fraction of points within the circle will be $\pi/4$, given a sufficient number of randomly selected samples.

Monte Carlo Methods (3)

Computing an Integral

One quadrant of the construction in Figure 3.7 can be described by the integral

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

A random pair of numbers, (x_r, y_r) would be generated, each between 0 and 1, and then counted as in circle if $y_r \le \sqrt{1-x_r^2}$; that is, $y_r^2 + x_r^2 \le 1$.

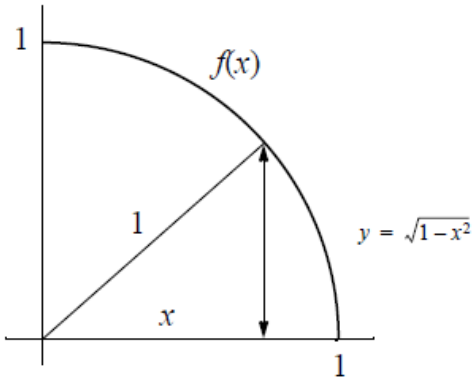


Figure 3.8 Function being integrated in computing π by a Monte Carlo method.

Monte Carlo Methods (4)

Alternative (better) Method

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

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

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

Monte Carlo Methods (5)

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);
```

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

Monte Carlo Methods (6)

Parallel Implementation

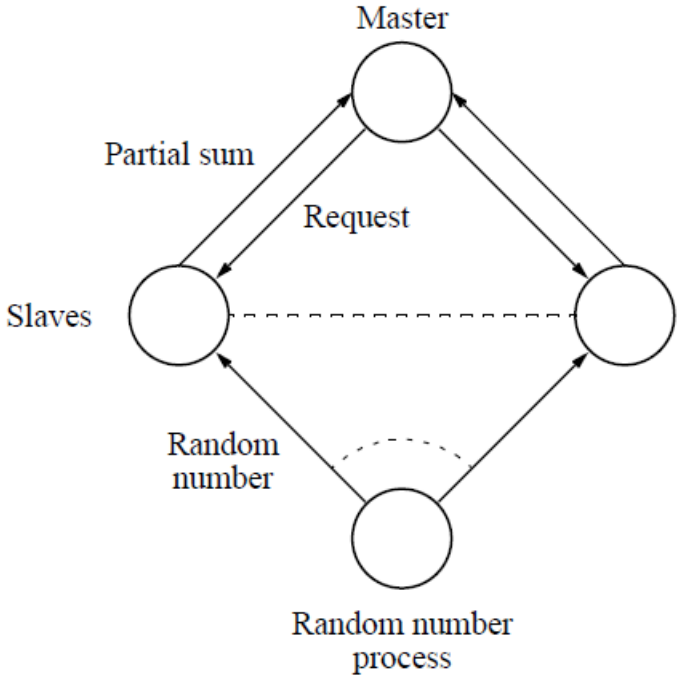


Figure 3.9 Parallel Monte Carlo integration.

Monte Carlo Methods (7)

Pseudocode

Master

```
for(i = 0; i < N/n; i++) {
    for (j = 0; j < n; j++)      /* n=no 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 < slave_no; i++) { /* terminate computation */
    recv(P_i, req_tag);
    send(P_i, stop_tag);
}
sum = 0;
reduce_add(&sum, P_group);
```

Monte Carlo Methods (8)

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);
```

Random Number Generation

The most popular way of creating a pueudorandom number sequence:

$$x_1, x_2, x_3, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{n-1}, x_n,$$

is by evaluating x_{i+1} from a carefully chosen function of x_i, often of the form

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

where a, c, and m are constants chosen to create a sequence that has similar properties to truly random sequences.

- A good generator is with $a = 16807$, $m = 2^{31} - 1$, and $c = 0$.
  - This generator creates a repeating sequence of $2^{31} - 2$ different numbers

Parallel Random Number Generation

It turns out that

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

$$x_{i+k} = (Ax_i + C) \bmod m$$

where $A = a^k \bmod m$, $C = c(a^{k-1} + a^{n-2} + \ldots + a^1 + a^0) \bmod m$, and k is a selected "jump" constant.

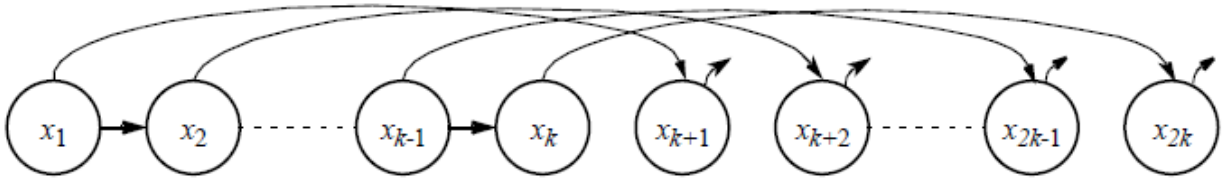


Figure 3.10 Parallel computation of a sequence.