Chapter 3 - Embarrassingly Parallel Computations

Embarrassingly (Pleasantly) Parallel Computation: "The ideal computation"

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

- **parallelizing these problems are obvious**
  - **no special techniques or algorithms: just distribute data & start processes**
- **no communication between the separate processes**
  - **each process need different data and produces results from its input w/o any need for results from other processes**
- **gives maximum speedup**

Often the independent parts are identical computations and SPMD model is appropriate

Data is not shared, but copied to each process if necessary

Nearly Embarrassingly Parallel Computation:

- requires results to be distributed and collected and combined in some way.
  - Initially and finally a single process must be operating alone
    - Master-Slave Organization (w/ dynamic or static process creation)
    - Even if the slave processes are all identical, we may not get the optimum solution if the processors are different.

Geometrical Transformations of Images

Displayed images on a computer often originate in two ways:

- Images obtained from external sources.
  - image processing
- Images that are artificially created
  - computer graphics

Graphical operations can be performed upon a stored image:

- move, resize, rotate on regular images
- smoothing and edge detection on noisy images

Pixmap: the most basic way to store a 2-D image in which each pixel (picture element) is stored as a binary number in a 2-D array.

B/W Images: bitmap (a single bit is is sufficient for each pixel)

Grayscale Images: 8 bits to represent 256 different monochrome intensities.

Color Images: Three primary colors, R/G/B, are stored as separate 8-bit numbers. ("tiff" format)

GIF file format:

storage requirements for color images can be reduced by using a look-up table to hold the RGB representation of specific colors used in the image.

256 different colors ⇒

- 256 24-bit entries could hold the representation of the colors used.
- Each pixel in the image store only 8-bits to select the appropriate table entry

(Look-up table is held in the file together with the image.)

Geometric Transformations

Xformations on each pixel is totally independent from the xformations on other pixels.

- result of a xformation is simply an updated bitmap.

- Shifting:

  $x' = x + \Delta x$

  $y' = y + \Delta y$
- Scaling: (enlarge if S>1, reduce if S<1)

  $x' = xS_x$

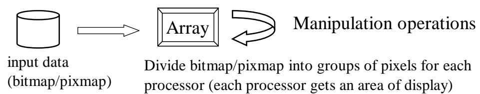
  $y' = yS_y$
- Rotation: (through an angle θ about the origin)

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

  $y' = -x \sin\theta + y \cos\theta$
- Clipping: x_l, y_l (x_h, y_h) is the lowest (highest) values of x, y in the area to be displayed:

  $x_l \le x' \le x_h$

  $y_l \le y' \le y_h$



input data (bitmap/pixmap)

Divide bitmap/pixmap into groups of pixels for each processor (each processor gets an area of display)

Two general methods for grouping:

- by square/rectangular regions
- by columns/rows

No effect on communication in Embar.Parallel Compn.

- Ex: With 640x480 image and 48 processors, how would you do the data distribution?

Master

```
for (i=0; row=0; i<48; i++, row+=10)
   send(row, P_i);                                  //send row number
for (i=0; i<480; i++)
   for (j=0; j<640; j++)
        temp_map[i][j] = 0;
for (i=0; i < (640*480); i++) {
   recv(oldrow, oldcol, newrow, newcol, P_ANY);   //accept new coords
   if !((newrow<0)||(newrow>=480) || (newrow<0)||(newrow>=640))
        temp_map[newrow][newcol]=map[oldrow][oldcol];
}
for (i=0; i<480; i++)
   for (j=0; j<640; j++)
        map[i][j] = temp_map[i][j];
```

Slave

```
recv(row, P_master);                                     //receive row number
for (oldrow=row; oldrow<(row+10); oldrow++)
   for (oldcol=0; oldcol<640; oldcol++) {                // transform coords
        newrow = oldrow + delta_x;
        newcol = oldcol + delta_y;
        send(oldrow, oldcol, newrow, newcol, P_master); //coords to master
   }
```


Analysis.

Each pixel requires 1 computational step, there are n x n pixels:

- $t_s = n^2 \Rightarrow O(n^2)$
- $t_p = t_{comm} + t_{comm}$

where $t_{comm} = t_{startup} + m\ t_{data} \Rightarrow O(m)$ with p processors,

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

$t_{comp} = 2(n^2/p) = O(n^2/p)$

Total t_p is $O(n^2)$

But $t_{startup}$ constant hidden in t_{comm} far exceeds those constants in the computation in most practical situations.

What can we do?

combine messages, send results back in groups.

Mandelbrot Set

Again a bit mapped image is manipulated, but this time it involves significant computation.

Mandelbrot Set is a set of points in complex plane that are quasi-stable when computed by iterating a function, such as $z_{k+1} = z_k^2 + c$, initial value for z is 0.

Iterations continue until magnitude of z > 2 or iteration number > a threshold value.

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

In each iteration,

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

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

Sequential Code:

```
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;
      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;
}
```


Obtain the actual complex plane coordinates by scaling:

```
c.real = real_min + x * (real_max  - real_min)/disp_height;
c.imag = imag_min + y * (imag_max - imag_min)/disp_width;
```

For computational efficiency:

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

Including scaling, the could could be of the form:

```
for (x=0; x<disp_width; x++)       /* screen coordinates x & 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);
}
```

See http://www.cs.uncc.edu/par_prog
uses Xlib calls for the graphics.

Parallellizing the Mandelbrot Set Computation

Each pixel can be computed w/o any info about the surrounding pixels.

We will try

- static task assignment
- dynamic task assignment

Static Task Assignment

- Each processor is assigned a fixed area of display.
- Grouping by square/rectangular regions or by columns/rows.

Suppose that a display area of 640x480 and we have 48 processes.

Static Task Assignment

Master

```
for (i=0; row=0; i<48; i++, row+=10)
    send(row, P_i);                               // send row number

for (i=0; i < (640*480); i++)  {
    recv(&c, &color, P_ANY);                      // accept coords/colors
    display(c, color);
}
```

Slave

```
recv(row, P_master);                              // receive row number
for (x=0; x<disp_width; x++)
    for (y=row; y<(row+10); y++)  {               // screen coordinates x and y
        c.real = real_min + ((float) x * scale_real);
        c.imag = imag_min + ((float) y * scale_ imag);
        send(&c, &color, P_master); // send coords & color to master
    }
```


Dynamic Task Assignment - Work Pool/Processor Farms

Mandelbrot Set requires significant iterative computation for each pixel:

- # iterations will generally be different for each pixel.
- computers may be of different type, or operate at different speeds.
- **Hence some processors may complete their assignment before others**

Ideally we want all processors to finish together, achieving a system efficiency of 100%, which can be addressed using load balancing.

Different sizes of regions could be assigned to different processors, but this would not be satisfactory:

- we may not a priori each processor's computational speed,
- we would have to know the exact time it takes for each processor to compute each pixel.

Workpool Approach to Dynamic Task Assignment

- Processors are supplied with work when they become idle
  - Sometimes called processor farm, when all processors are the same type.
- Workpool holds a collection, pool, of tasks to be performed.
  - in our case, the set of pixels forms the tasks
  - when a processor has computed the color for the pixel, it returns the color and requests a further pair of pixel coordinates from the work pool
  - when all pixel coordinates have been taken, we than have to wait for all the processors to complete and report in for more pixel coordinates.

  Sending pairs of coordinates of individual pixels will result in excessive communication ⇒ group them.

  In workpool solution, some pixels will be generated before others.

Master

```
count = 0;
row = 0;
for (k=0; k<procno; k++) {
  send(&row, P_k, data_tag); //send row #
  count++;
  row++;
}
do {
  recv(&slave,&r,color,P_ANY,result_tag);
  count--;
  if (row < disp_height) {
    send(&row, P_slave, data_tag);
    row++;
    count++;
  } else
    send(&row, P_slave, terminator_tag);
    rows_recv++;
    display(r, color);
} 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++)  {
    c.real = real_min + ((float)x*scale_real);
    color[x] = cal_pixel(c );
  }                           // send row colors to master
  send(&i,&color,P_master,result_tag);
  recv(y, P_master, source_tag);
}
```


Analysis.

We don't know how many iterations are needed for each pixel. We only know that # iterations for each pixel is some function of *n* but cannot exceed *max*.

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

Phase 1: Communication.

$t_{comm1} = s(t_{startup} + t_{data})$ where s is the number of slaves.

Phase 2: Computation.

$t_{comp} \leq (max \times n) / s$

Phase 3: Communication.

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

Overall $t_p = (max \times n) / s + (n/s + x)(t_{startup} + t_{data})$

Monte Carlo Methods

The basis of Monte Carlo methods is the use of random selections in calculations that lead to the solution to numerical and physical problems.

Example: Calculating Π number.

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

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

The area of any shape within a known bound area could be computed by the preceeding method, or any area under a curve; i.e., an integral.

- MC methods are would not be used in practice for 1-D integrals, for which quadrature methods are better. They would be very useful for integrals with a large number of variables.

Sequential Code: **for computing $f(x) = x^2 - 3x$**

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

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


Parallel Random Number Generation

- The most popular way of creating a pseudorandom number sequence is by evaluating x_{i+1} from a carefully chosen function of x_i.
  - The key is to find a function that will create a very large sequence with the correct statistical properties:

    $x_{i+1} = (a\, x_i + c) \bmod m$ *Linear Congruential Generator*

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

Even though it appears that the pseudorandom number computation is sequential in nature, as each number is calculated from the previous number, a parallel formulation is possible.

$$x_{i+1} = (a\, x_i + c) \bmod m$$

$$x_{i+k} = (A\, x_i + C) \bmod m$$

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