*

Embarrassingly Parallel Computations

*Material based on B. Wilkinson et al., "PARALLEL PROGRAMMING. Techniques and Applications Using Networked Workstations and Parallel Computers"

Ideal Parallel Computation

Parallel programs divide a problem into parts that can be run concurrently on multiple processors.

A problem that can be divided right away into 100% independent parts that can be executed in parallel is an ideal parallel computation.

Such a beatific problem is called *embarrassingly parallel.*

Parallelizing them should be obvious. No special technique or algorithm needed.

Only need means to distribute and collect data and start the processes.

A true embarrassingly parallel problem should require no communication between separate processes. See Figure 3.1.

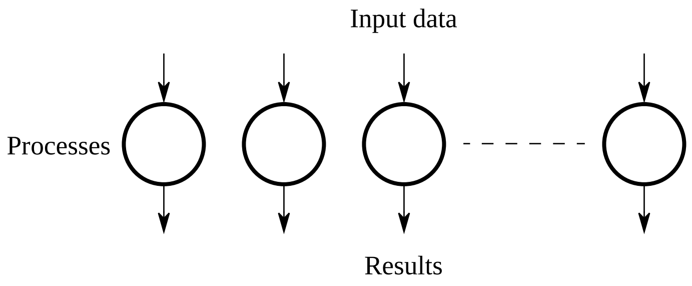


Figure 3.1 Disconnected computational graph (embarrassingly parallel problem).

Ideal Parallel Computation Cont.

Each process needs different (or same) set of data and can produce its results without need of the results of the other processes.

Gives maximum speedup if all processors can be kept busy for entire computation.

Often, independent parts are the same computation, so can use SPMD model.

Problems that require results to be distributed, collected, and combined in some manner are called *nearly embarrassingly parallel computations.*

This is when you have a master process sending out data to slaves. The slaves process data, then send it back to master who stores it, or combines the results.

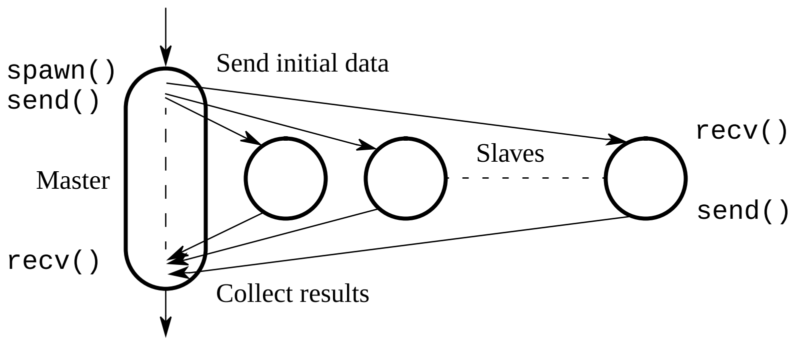


Figure 3.2 Practical embarrassingly parallel computational graph with dynamic process creation and the master-slave approach.

Load Balancing

Even with identical slave processes, if our assignment of processes to processors is fixed, we may get suboptimal performance.

Will need to use load-balancing methods to improve performance.

Will introduce in this chapter, but only for non-interacting slave processes.

Examples of Embarrassingly Parallel Problems

Geometrical Transformation of Images: Two dimensional images are stored in a *pixmap.* For each pixel in the array, a number is stored to represent it.

Common to perform a geometrical transformation on image, such as *shifting*, *scaling*, *rotation*, *clipping*, etc.

Need to perform mathematical operations on each pixel. Transformation of each pixel is independent of the transformation on the other pixels.

Mandelbrot Set

To display the mandelbrot set we again have to process a bit-mapped image.

This time, we have to first calculate it!

A Mandebrot set is a group of points in the complex plane that is quasi-stable when calculated by iterating some function. Normally, use:

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

Will create sequence: $z_0 = 0, z_1, z_2, \ldots$

The number c represents a point on the complex plane.

Complex Number Refresher

A complex number is:

$$z = a + bi = z_{\text{real}} + z_{\text{imag}} i \quad \text{where } i = \sqrt{-1}.$$

The magnitude of z is: $z_{\text{magn}} = \sqrt{a^2 + b^2}$

Examining equation, $z_{k+1} = z_k^2 + c$, we see we can simplify the computation by noting that:

$$\begin{aligned} z^2 &= a^2 + 2abi + (bi)^2 \\ &= a^2 - b^2 + 2abi \end{aligned}$$

We can thus compute z_{k+1} with $z_k = z_{\text{real}} + z_{\text{imag}} i$ as follows:

$$\begin{aligned} z_{k+1}.\text{real} &= z_{\text{real}}^2 - z_{\text{imag}}^2 + c_{\text{real}} \\ z_{k+1}.\text{imag} &= 2 z_{\text{real}} z_{\text{imag}} + c_{\text{imag}} \end{aligned}$$

Calculating Set

The value of c ranges over the complex plane from c_{min} to c_{max}.

If we treat $c = x + yi = (x, y)$ as a tuple, then we can take the coordinates (x, y) as a point on the x-y plane. See Figure 3.4.

This means that c_{min}.real $\leq c_{\text{real}} \leq c_{\text{max}}$.real and c_{min}.imag $\leq c_{\text{imag}} \leq c_{\text{max}}$.imag.

For each value of c in our range, we compute the *color* of that point as an 8-bit number (0-255).

We calulate the color of c by computing z_{k+1} until the magnitude exceeds two ($z_{\text{magn}} = \sqrt{a^2 + b^2} \geq 2$) or the number of iteration reaches some constant (for us, 255).

The number of iterations is the color of c.

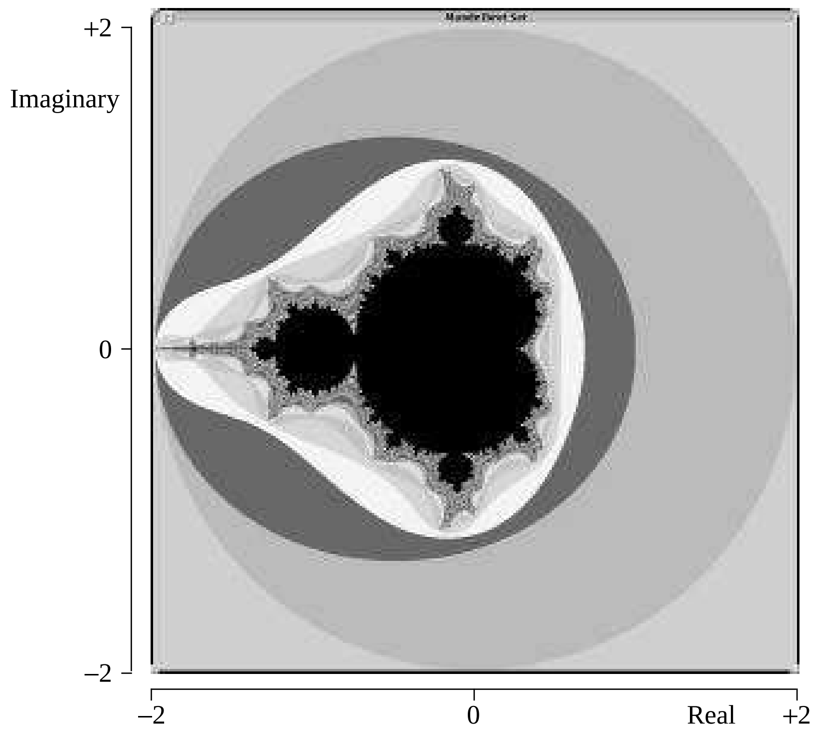


Figure 3.4 Mandelbrot set.

Sequential Code

Represent the complex number using the structure:

```
structure complex {
   float  real;
   float  imag;
};
```

A routine for calculating z for point c and returning the color would be similar to:*

```
int cal_pixel(complex c){

  int count, max;
  complex z;
  float temp, lengthsq;
```

*This and other code from "Parallel Programming.. " by Wilkinson et al.

Sequential Code Cont.

```
  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;  /* the color of point c  */
}
```

Stops when $\sqrt{a^2 + b^2} \geq 2$ or maximum count reached.

Means all the Mandelbrot points must be within a circle centered at the origin, with radius 2.

Scaling the Coordinate System

Want to display image on a display of fixed size: *disp_height* $\times$ *disp_width* in pixels.

This creates a rectangular window that can be positioned anywhere in the complex plane.

We need to map each pixel onto the complex plane to determine a corresponding value for c.

We first choose values for our range: $c_{\text{min}} = (\text{real\_min}, \text{imag\_min})$ to $c_{\text{max}} = (\text{real\_max}, \text{imag\_max})$.

Scaling the Coordinate System cont.

We will thus need to scale each (x, y) value from our display to get our value for c.

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

To improve speed, we define:

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

To process every point in the display, we'd use:

```
for (x = 0; x < disp_width; x++)
  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 suitable routine to display the pixel (x,y) with the indicated color.

Parallelizing Computation

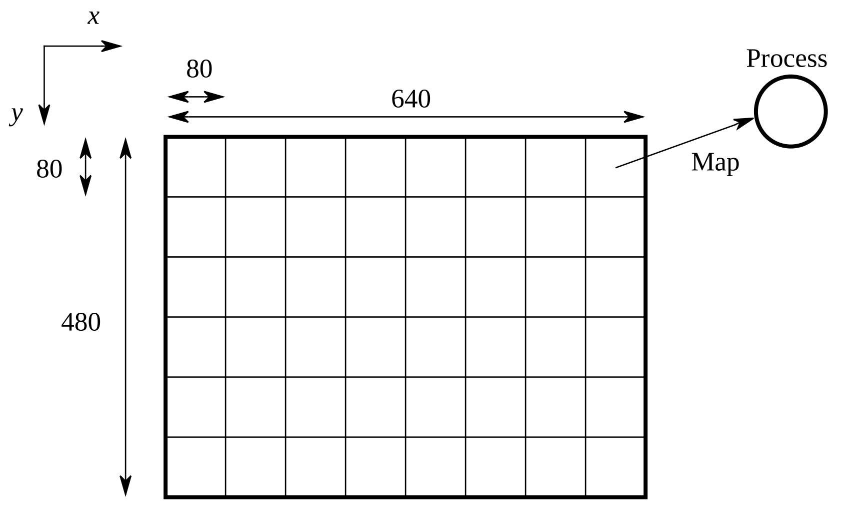
Parallelizing works very well for message-passing systems as value for each pixel can be calculated without requiring information about neighboring pixels.

Will first examine a *static task assignment.*

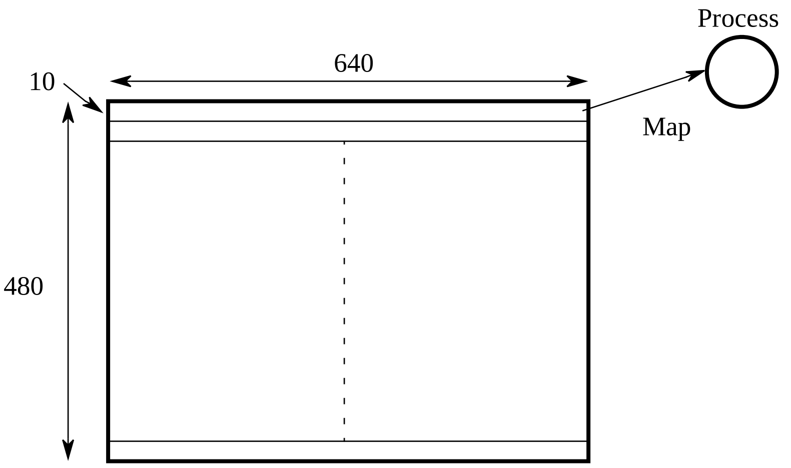
Can do this by breaking up the display area into squares or rectangles and assign each to a process. See Figure 3.3.

Once process assigned portion of display, it must call *cal_pixel()* to process every pixel in its area.

Suppose display is 640×480 and we want to compute 10 rows per process. We thus need 49 processes.



(a) Square region for each process



(b) Row region for each process

Figure 3.3 Partitioning into regions for individual processes.

Parallelizing Computation Cont.

Could use following pseudocode:

```
   Master Code:

for (i = 0, row = 0; i <48; i++, row = row + 10)
   send(&row, Pi)  /* send starting row to proc i */

/* receive pixels individually from any process */
for (i = 0; i < (480 * 640); i++) {
  /* receive coordinates/ color from any process*/
  recv(&c, &color, Pany);
  display(c,color);
}

  Slave Code (process i):

/* receive starting row number */
recv(&row,Pmaster);
for (x = 0; x < disp_width; x++)
  for (y = row; y < (row + 10); y++)  {
    c.real = real_min  + ((float)x * scale_real);
    c.imag = imag_min  + ((float)y * scale_imag);
    color = cal_pixel(c);
    /* send coordinates/color of pixel to master */
    send(&c,&color,Pmaster);
  }
```

Problems With Code

- Obviously, won't perform optimally if processors are not the same. Also, computation time of each pixel varies, so want dynamic allocation.

- Should have partitioned data based on number of processes, not on a fixed process number (48 slaves + 1 master) (see sample collective MPI program given earlier).

- Pixels are sent to master one at a time.

  If using message passing, this incurs significant startup overhead.

  Better to save to array, and send results one row at a time.

- Slave routine should return (x, y) coordinates, not c.

Dynamic Task Assignment

Calculating Mandelbrot set requires significant computation.

Number of iterations per pixel normally different plus computers used may be different types or operate at different speeds.

Thus, some processors might complete their tasks before others, and thus be idle.

Will use *load balancing* so that all processors finish together.

Varying Portion Size: Can assign different sized regions to different processors.

Undesirable as we would need to know in advance the speed of each processor as well as the time to calculate each Pixel.

Work Pool/Processor Farms

More efficient approach is a dynamic form of load balancing such as the *work pool* approach.

Basic idea: supply individual processors with chunks of data when they become idle.

The work pool holds a collection (pool) of tasks to be performed.

For our application, the coordinates of our set of pixels is our set of tasks.

The number of tasks is fixed as the number of pixels is fixed before computation begins.

How Work Pool Functions.

Our work pool contains coordinates for rows of pixels.

Requires that the slave processes are informed at the begining the number of pixels in a row.

Individual processes request a row, and then process it.

Process then returns the array of colurs for row, and requests new row.

When all rows have been assigned, we then wait for each process to finish their tasks, and return the data.

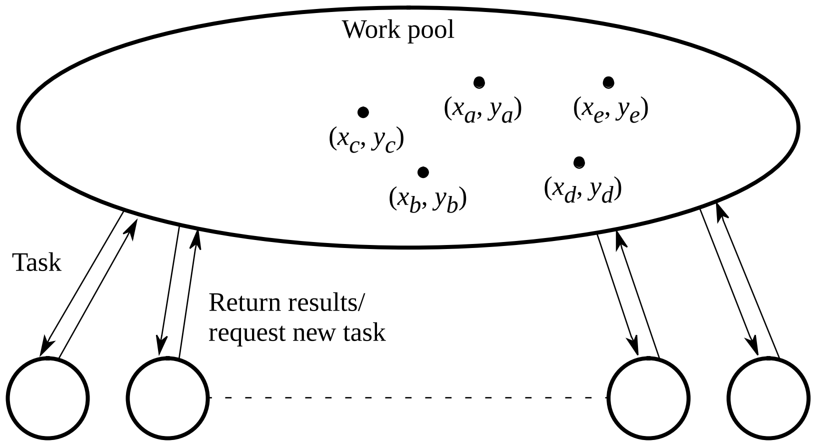


Figure 3.5 Work pool approach.

Work Pool Pseudocode

Assuming *procno* is the number of slave processes, we could use the following pseudocode:

```
   Master Code:

count = 0;  /* termination counter */
row = 0;    /* row being sent */
rows_recv = 0; /* No of rows processed */

/* assuming procno < disp_height */
for (k = 0; k < procno; k++) {
  /* send initial row to process */
  send(&row, P_k, data_tag);
  count++;
  row++;
}

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

Work Pool Pseudocode Cont.

```
  Slave Code (process i):

/* receive 1st row  */
recv(&y, P_master, ANYTAG, source_tag);

/* exit when a tag other than data_tag received */
while (source_tag == data_tag) {
  c.imag = imag_min  + ((float)y * scale_imag);

  /* compute colours for each pixel in row */
  for (x = 0; x < disp_width; x++)
    c.real = real_min  + ((float)x * scale_real);
    color[x] = cal_pixel(c);
  }

  /* send row no/color of pixels to master */
  send(&i, &y, color, P_master, result_tag);
  /* receive next row to process */
  recv(&y, P_master, ANYTAG, source_tag);
}
```

Discussion of Pseudocode

As rows assigned on dynamic basis, will easily handle differences in computation time.

Termination of main loop in master based on number of rows outstanding as shown in Figure 3.6.

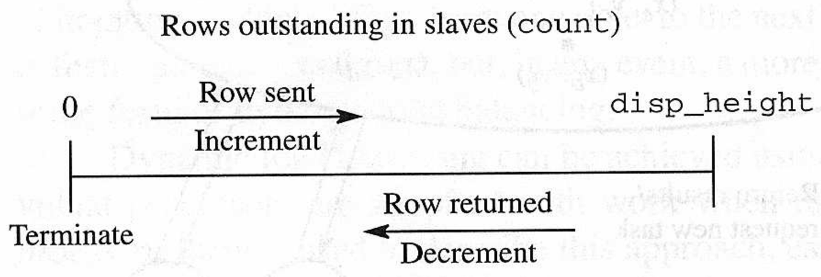


Figure 3.6 Counter termination. *

Alternate termination condition is to count number of rows received. When *rows_recv* = *disp_height*, then exit loop.

*From "Parallel Programming.. " by Wilkinson et al.

Tradeoff for Work Pool Method

Work pool method allows for dynamic work load balancing providing higher CPU utilization.

Increases number of messages sent thus cost of communication.

Need to find balance between size of task, versus number of tasks.

Smaller task (data chunk) size decreases computation: communication ratio.

Timing Analysis of Mandelbrot Set

Exact analysis not easy as don't know how many iterations would be needed for each pixel.

For n pixels, the number of iterations for a given pixel is a function of n, but can't exceed value *max*.

This gives us formula for the sequential time:

$$t_s \leq \mathit{max} \times n$$

In other words, time complexity of $O(n)$.

Parallel Timing Analysis

We will examine for the static assignment case.

We have three main phases: communication, computation, and more communication.

Phase 1: Communication. Send starting row number to each of s slaves.

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

Phase 2: Computation. Slaves perform computation for mandelbrot in parallel.

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

Parallel Timing Analysis Cont.

Phase 3: Communication. Results are sent to master one pixel at a time.

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

The total parallel time is:

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

This gives us an estimate for our scaling factor of:

$$S(n) = \frac{\mathit{max} \times n}{\frac{\mathit{max} \times n}{s} + (\frac{n}{s} + s)(t_{\text{startup}} + t_{\text{data}})}$$

For large values of *max*, this gives $S(n) = p - 1$, where $p = s + 1$ is the total number of processes.