Embarrassingly Parallel Computations

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

Figure 3.2: Disconnected computational graph (embarrassingly parallel problem).
Geometrical Transformations of Images

Examples of low level embarrassingly parallel image operations:

(a) Shifting
The coordinates of an object shifted by a distance $D_x$ in the $x$-dimension and $D_y$ in the $y$-dimension are given by

\[
\begin{align*}
    x' &= x + D_x \\
y' &= y + D_y
\end{align*}
\]

(b) Scaling
The coordinates of an object scaled by a factor $S_x$ in the $x$-direction and $S_y$ in the $y$-direction are given by

\[
\begin{align*}
    x' &= xS_x \\
y' &= yS_y
\end{align*}
\]

The object is enlarged in size when $S_x$ and $S_y$ are greater than 1 and reduced in size when $S_x$ and $S_y$ are between 0 and 1. Note that the magnification or reduction do not need to be the same in both $x$- and $y$-directions.

(c) Rotation
The coordinates of an object rotated through an angle $\theta$ about the origin of the coordinate system are given by

\[
\begin{align*}
    x' &= x\cos\theta - y\sin\theta \\
y' &= x\sin\theta + y\cos\theta
\end{align*}
\]

Examples of low level embarrassingly parallel image operations:

- Shifting
- Scaling
- Rotation

Parallel Programming concern is division of bitmap/pixmap into groups of pixels for each processor because there are usually many more pixels than processors. Two general methods of grouping are by square/rectangular regions and by columns/rows.
Pseudocode to Perform Image Shift

**Master**

```plaintext
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**

```plaintext
recv(row, Pmaster); /* 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, Pmaster); /* coords to master */
    }
```

---

**Analysis**

Suppose each pixel requires one computational step and there are \( n \times n \) pixels.

**Parallel**

\[
\text{Computation} = O\left(\frac{n^2}{p}\right)
\]

\[
\text{Parallel Communication} = t_{\text{startup}} + 2t_{\text{data}} + 4n^2(t_{\text{startup}} + t_{\text{data}}) = O(n^2 + pn^2)
\]

**Sequential**

\[
\text{Sequential Computation} = n^2
\]

\[
\text{Sequential Parallelism} = \frac{n^2}{p}
\]

However, the constant hidden in the communication part far exceeds those constants in the computation part.

**Overall Execution Time**

\[
t_p = t_{\text{computation}} + t_{\text{communication}}
\]

For constant \( p \), this is \( O(n^2) \).
Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen
Prentice Hall, 1999

Mandelbrot Set

A set of points in the 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 \( z_k \) 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.

The iterations are continued until the magnitude of \( z \) is greater than 2 or the number of iterations reaches some arbitrary limit.

The magnitude of \( z \) is the length of the vector given by

\[ |z| = \sqrt{a^2 + b^2} \]

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

\[ z^2 = (a + bi)^2 = a^2 + 2abi + b^2 \]

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:

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

Sequential Code

Routine for computing value of one point and returning number of iterations for real and imaginary parts of \( z \):

```c
struct 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;
}
```

The length of \( z \) is the length of the vector given by

\[ |z| = \sqrt{a^2 + b^2} \]

The length of \( z \) is used in some computations. The magnitude of \( z \) is greater than 2 or the number of iterations reaches some arbitrary limit.

The initial value for the function \( f(z) = z^2 + c \) 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 the function \( f(z) = z^2 + c \) 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.
Suppose the display height is $disp_{height}$, the display width is $disp_{width}$, and the point in this display area is $(x, y)$.

For computational efficiency, let

$$scale_{real} = \frac{real_{max} - real_{min}}{disp_{width}};$$
$$scale_{imag} = \frac{imag_{max} - imag_{min}}{disp_{height}};$$

Including scaling, the code could be of the form

```c
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 suitable within to display the pixel $(x, y)$ at the computed color.
Parallelizing Mandelbrot Set Computation

Static Task Assignment

Dynamic Task Assignment

WORK POOL/PROCESSOR PArts

Parallelizing Mandelbrot Set Computation
Coding for Work Pool Approach

```c
// Master
count = 0; /* counter for termination*/
row = 0; /* row being sent */
for (k = 0; k < procno; k++) { /* assuming procno<disp_height */
    send(&row, P, 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, data_tag); /* send next row */
        row++; /* next row */
        count++;
    } else
        send (&row, P, terminator_tag); /* terminate */
    rows_recv++;
display (r, color); /* display row */
} while (count > 0);
```

Slave
recv(y, Pmaster, 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, Pmaster, source_tag); /* receive next row */
};

Figure 3.6
Counter termination. Terminate

<table>
<thead>
<tr>
<th>Count</th>
<th>Increment</th>
<th>Decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>0</td>
<td>Rows outstanding in slaves (count)</td>
</tr>
<tr>
<td>Count</td>
<td>0</td>
<td>Rows outstanding in slaves (count)</td>
</tr>
</tbody>
</table>
Sequentially, the fraction of points within the circle will be \( \frac{\text{area of circle}}{\text{area of square}} \). Points within the square are chosen randomly and a score is kept of how many points happen to be within the circle. The ratio of the area of the circle to the square is given by

\[
\frac{\text{area of circle}}{\text{area of square}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}.
\]

**Example - To Calculate \( \pi \)**

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

**Analysis**
Figure 3.8 Function being integrated in computing $p$ by a Monte Carlo method.

Computing an Integral

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

$$\int_{x_r}^{1} \int_{y_r}^{1} f(x) \, dx \, dy$$

A random pair of numbers, $(x_r, y_r)$, would be generated, each between 0 and 1, and then counted as in circle if $x^2 + y^2 \leq 1$, that is, 

$$x^2 + y^2 \leq 1$$

Alternatively (better) Method

The routine randv(xl, x2) returns a pseudorandom number between $x_1$ and $x_2$.

The sequential code might be of the form:

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

The routine randv(x1, x2) returns a pseudorandom number between $x_1$ and $x_2$.

**Example**

Compute the integral of $f(x)$ from $x_1$ to $x_2$.

An alternative probabilistic method is to use the random values of $x$ to compute the integral:

$$\int_{x_1}^{x_2} f(x) \, dx$$

One quadrant of the construction in Figure 3.7 can be described by the integral $x^2 + y^2 \leq 1$. Compute $f(x)$ by integrating $f(x)$ over $x$ from $x_1$ to $x_2$. The sequential code is as follows:

```
sum = 0;
for (i = 0; i < N; i++) { /* N random samples */
  x_r = randv(x1, x2); /* generate next random value */
  y_r = randv(x1, x2); /* generate next random value */
  if ($x_r^2 + y_r^2 \leq 1$) sum = sum + $x_r^2$ - 3 * $x_r$; /* compute f(x_r) */
}
area = (sum / N) * (x2 - x1);
```

The routine randv(x1, x2) returns a pseudorandom number between $x_1$ and $x_2$. The area under the mean function $f(x)$, integrated over $x$, can be determined as:

$$\int_{x_1}^{x_2} f(x) \, dx$$

This is an alternative probabilistic method to find an integral.
Parallel Monte Carlo integration.

**Pseudocode**

```c
// 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(PANY, req_tag, Psource); /* wait for a slave to make request */
    send(xr, &n, Psource, compute_tag);
}
for (i = 0; i < slave_no; i++) /* terminate computation */
    recv(Pi, req_tag);
send(Pi, stop_tag);
sum = 0; reduce_add(&sum, Pgroup);

// Slave
sum = 0; send(Pmaster, req_tag);
recv(xr, &n, Pmaster, source_tag);
while (source_tag == compute_tag) {
    for (i = 0; i < n; i++)
        sum = sum + xr[i] * xr[i] - 3 * xr[i];
    send(Pmaster, req_tag);
    recv(xr, &n, Pmaster, source_tag);
}
reduce_add(&sum, Pgroup);
```
Parallel Random Number Generation

The most popular way of creating a pseudorandom 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

\[
x_{i+1} = (ax_i + c) \mod m
\]

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

**Parallel Pseudorandom Number Generators**

The most popular way of creating a pseudorandom number sequence is by evaluating \( x_{i+1} \) from a carefully chosen function of \( x_i \), often of the form

\[
x_{i+1} = (ax_i + c) \mod m
\]

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