Embarrassingly Parallel Computations

Embarrassingly Parallel Computations

- A computation that can be divided into completely independent parts, each of which can be executed on a separate process(or) is called embarrassingly parallel.
- An embarrassingly parallel computation requires none or very little communication.
- A nearly embarrassingly parallel is an embarrassingly parallel computation that requires initial data to be distributed and final results to be collected in some way. In between the processes can execute their task without any communication.

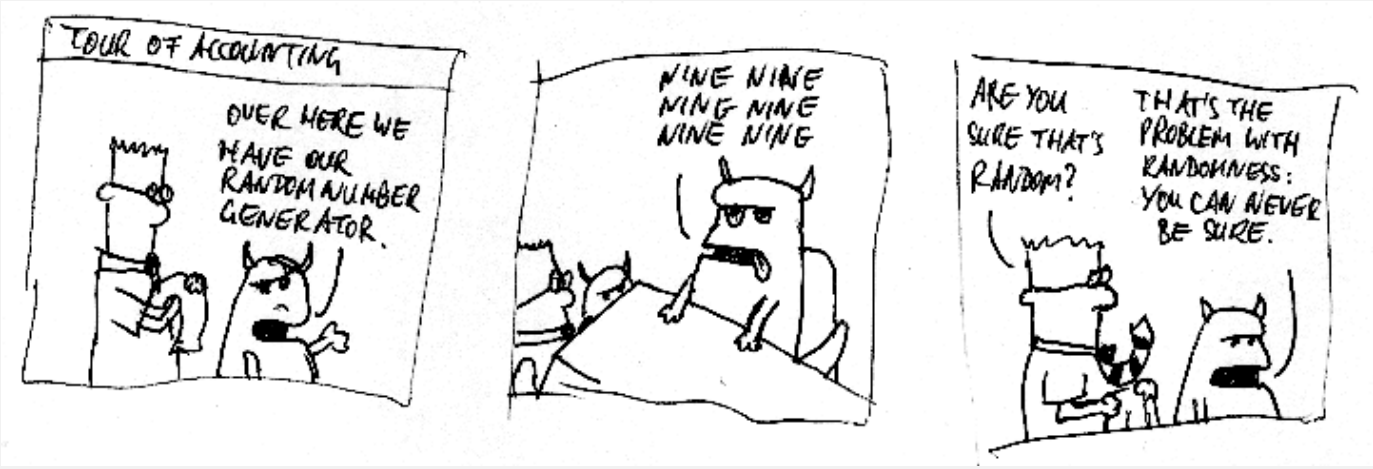
Embarrassingly Parallel Computation Examples

- Folding@home project: Protein folding software that can run on any computer with each machine doing a small piece of the work.
- SETI project (Search for Extra-Terrestrial Intelligence) `http://setiathome.ssl.berkeley.edu/`.
- Generation of All Subsets.
- Generation of Pseudo-random numbers.
- Monte Carlo Simulations.
- Mandelbrot Sets (a.k.a. Fractals)
- and many more

Generation of Pseudo-Random Numbers

Random number generators use a type of recurrence equation to generate a "reproducible" sequence of numbers that "appear random."

- By appear random we mean that they will pass statistical tests. By "reproducible" means that we will get the same sequence of pseudo-random numbers if we use the same starting number (the seed).



- The pseudo-random number generators are based on a linear congruential recurrence of the following form, where s is the initial seed value and c is a constant usually chosen based on its mathematical properties.

$$\begin{aligned} y_0 &= s \\ y_i &= ay_{i-1} + c, 1 \leq i \leq n-1 \end{aligned}$$

Parallel Generation of Pseudo-Random Numbers

- Technique 1. One process can generate the pseudo-random numbers and send to other processes that need the random numbers. This is sequential. The advantage is that the parallel program has the same sequence of pseudo-random numbers as the sequential program would have, which is important for verification and comparison of results.
- Technique 2. Use a separate pseudo-random number generator on each process. Each process must use a different seed. The choice of seeds used at each process is important. Simply using the process id or time of the day can yield less than desirable distributions. A better choice would be to use the `/dev/random` device driver in Linux to get truly random seeds based on hardware noise.
- Technique 3. Convert the linear congruential generator such that each process only produces its share of random numbers. This way we have parallel generation as well as reproducibility.

Converting the Linear Congruential Recurrence

Assume: p processors, the pseudo-random numbers generated sequentially are $y_0, y_1, \ldots, y_{n-1}$.
The Idea: Instead of generating the next number from the previous random number, can we jump by p steps to get to y_{i+p} from y_i?
Let us play a little bit with the recurrence.

$$
\begin{aligned}
y_0 &= s \\
y_1 &= ay_0 + c = as + c \\
y_2 &= ay_1 + c = a(as + c) + c = a^2 s + ac + c \\
y_3 &= ay_2 + c = a(a^2 s + ac + c) + c = a^3 s + a^2 c + ac + c \\
&\ldots \\
y_k &= a^k s + (a^{k-1} + a^{k-2} + \ldots + a^1 + a^0)c \\
y_k &= a^k y_0 + (a^{k-1} + a^{k-2} + \ldots + a^1 + a^0)c \\
y_{k+1} &= a^k y_1 + (a^{k-1} + \ldots + a^1 + a^0)c
\end{aligned}
$$

Now we can express y_{i+k} in terms of y_i as follows.

$$
y_{k+i} = \overbrace{a^k}^{A'} y_i + \overbrace{(a^{k-1} + a^{k-2} + \ldots + a^1 + a^0)c}^{C'} = A' y_i + C'
$$

Converting the Linear Congruential Recurrence (contd)

We finally have a new recurrence that allows us to jump k steps at a time in the recurrence. Setting $k = p$, for p processes, we obtain:

$$y_{i+p} = A' y_i + C'$$

To run this in parallel, we need the following:

- We need to precompute the constants A' and C'.
- Using the serial recurrence, we need to generate y_i on the ith process, $0 \leq i \leq p-1$. These will serve as initial values for the processes.
- We need to make sure that each process terminates its sequence at the right place.

Then each process can generate its share of random numbers independently. No communication is required during the generation. Here is what the processes end up generating:

Process P_0:	$y_0, y_p, y_{2p}, \cdots$
Process P_1:	$y_1, y_{p+1}, y_{2p+1}, \cdots$
Process P_2:	$y_2, y_{p+2}, y_{2p+2}, \cdots$
$\cdots$	$\cdots$
Process P_{p-1}:	$y_{p-1}, y_{2p-1}, y_{3p-1}, \cdots$

Parallel Random Number Algorithm

An SPMD style pseudo-code for the parallel random number generator.

prandom(i,n)
//generate n total pseudo-random numbers
//pseudo-code for the ith process, $0 \leq i \leq p-1$
//serial recurrence $y_i = ay_{i-1} + c$, $y_0 = s$

1. compute y_i using the serial recurrence
2. compute $A^{'} = a^p$
3. compute $C^{'} = (a^{p-1} + a^{p-2} + \ldots + 1)c$

4. **for** (j=i; j<n-p; j=j+p)
5. $\quad y_{j+p} = A^{'} y_j + C^{'}$;
6. $\quad$ process y_j

GNU Standard C Library: Random Number Generator

```
#include <stdlib.h>

long int random(void);
void srandom(unsigned int seed);
char *initstate(unsigned int seed, char *state, size_t n);
char *setstate(char *state);
```

- The GNU standard C library's random() function uses a linear congruential generator if less than 32 bytes of information is available for storing the state and uses a lagged Fibonacci generator otherwise.
- The initstate() function allows a state array state to be initialized for use by random(). The size of the state array is used by initstate() to decide how sophisticated a random number generator it should use - the larger the state array, the better the random numbers will be. The seed is the seed for the initialization, which specifies a starting point for the random number sequence, and provides for restarting at the same point.

PRAND: A Parallel Random Number Generator

- Suppose a serial process calls random() 50 times receiving the random numbers....

```
SERIAL:
abcefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
```

  Each process calls random() 10 times receiving the values...

```
process 0:  abcdefghij
 process 1:  klmnopqrst
 process 2:  uvwxyzABCD
 process 3:  EFGHIJKLMN
 process 4:  OPQRSTUVWX
```

  Leapfrog parallelization is not currently supported. Neither is independent sequence parallelization.

- The principal function used for this is called `unrankRand()`. The function `unrankRand()` permutes the state so that each process effectively starts with its random() function such that the numbers it generates correspond to its block of random numbers. It takes a parameter called stride that represents how many random() calls from the current state the user want to simulate. In other words the following code snippets are functionally equivalent (although unrankRand() is faster)

```
ITERATIVE
for(i=0;i<1000000;i++)
    random();
USING UNRANK
unrankRand(1000000);
```

Using the PRAND library

- The header file is `prand.h` and the linker option is `-lprand`. The linker option needs to be added to the `Makefile` file.
- To use the prand library in a parallel program we would use the following format:

```
SERIAL:
    srandom(SEED);    //Consume the whole range of random numbers
    for (i=0;i<n;i++) {
        tmp = random();
    ...
PARALLEL:         //Each process uses a fraction of the total range...
    srandom(SEED)
    unrankRand( myProcessID * (n/numberOfProcessors) );
    for  (i=0;i < (n/numberOfProcessors);i++) {
        tmp = random();
    ...
```

  The above code must be fixed if n does not divide evenly by the number of processors.

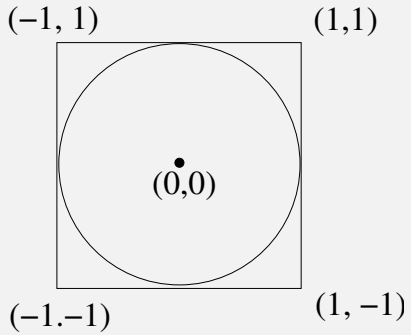
- See example: MPI/random/random.c. The PRAND library was developed by Jason Main, Ligia Nitu, Amit Jain and Lewis Hall. (`http://cs.boisestate.edu/~amit/research/prand/`)

Monte Carlo Simulations

- A *Monte Carlo simulation* uses random numbers to model a process. Monte Carlo simulations are named after the casinos in Monte Carlo. Monte Carlo approach.
  - Define a domain of possible inputs.
  - Generate inputs randomly from the domain using a chosen probability distribution.
  - Perform a deterministic computation using the generated inputs.
  - Aggregate the results of the individual computations into the final result.
- Monte Carlo methods are used in computational physics and related fields, graphics, video games, architecture, design, computer generated films, special effects in cinema, business, economics and other fields. They are useful when computing the exact result is infeasible.
- A large fraction of CPU time on some of the largest supercomputers is spent running Monte Carlo simulations for various problems.

A Monte Carlo Algorithm for Computing Pi

Consider a square of side of length 2 in the x coordinate in the range [-1,1] and y coordinate in the range [-1,1]. Now imagine a circle with unit radius with center at (0,0) inside this square. The area of this circle is $\pi r^2 = \pi$. The area of the enclosing square is $2 \times 2 = 4$. Hence the ratio is $\pi/4$.



The Monte Carlo method generates n points randomly inside the square. Suppose a random point has coordinates (x, y). Then this points lies inside the circle is $x^2 + y^2 < 1$. The method keeps track of how many points were inside the circle out of n total points. This ratio equals $\pi/4$ in the limiting case and allows us to calculate the value of π.

Parallel Monte Carlo Algorithm for Computing π

estimate_pi(n, p, seed)
//p processes, process number id is $0 \leq id \leq p-1$
//Assume that n is divisible by p
1. share = n/p
2. **for** (i=0; i<share; i++)
3. generate two random numbers x and y in the range [-1, 1]
4. **if** $(x^2 + y^2) < 1$
5. points = points + 1

6. **if** $(id == 0)$
7. Receive $p-1$ point counts from other processes
8. Add all the point counts
9. Calculate π = total point count/$n * 4$
10. **else**
11. Send my point count to P_0

Generation of Combinatorial Objects

Generation of combinatorial objects in a specific ordering is useful for testing.

- All subsets of a set (direct, graycode, lexicographic ordering).
- All permutations of a set.
- All k-subsets of an n-set.
- Composition of an integer n into k parts.
- Partition of an integer n
- and others

Generation of All Subsets in Direct Ordering

- Let the elements be $U = 1, 2, \ldots, n$.
- A subset S has rank m, where

$$m = a_1 + a_2 2 + a_3 2^2 + \ldots + a_n 2^{n-1}$$

  where $a_i = 1$ if $i \in S$ and 0 if $i \notin S$. We have 2^n possible subsets.
- A simple algorithm to generate all subsets is to map the subsets to the numbers $0 \ldots 2^n - 1$ and convert the number to binary and use that as a set representation.

Example of direct ordering of subsets

Rank	Binary	Subset
0	0 0 0	{}
1	0 0 1	{3}
2	0 1 0	[2}
3	0 1 1	{2,3}
4	1 0 0	{1}
5	1 0 1	{1,3}
6	1 1 0	{1,2}
7	1 1 1	{1,2,3}

Example of Lexicographic ordering of subsets

Rank	Binary	Subset
0	0 0 0	{}
1	1 0 0	{1}
2	0 1 0	[2}
3	0 0 1	{3}
4	1 1 0	{1,2}
5	1 0 1	{1,3}
6	0 1 1	{2,3}
7	1 1 1	{1,2,3}

Example of Gray code ordering of subsets

- To create a n-bit Gray code, take a $n-1$ bit Gray code and reflect it and then add 0 in front the codes on top and 1 in front of the codes in the bottom.
- In a Gray code, only one bit changes as we go from one subset to the next in the ordering. Similar to node numbering in hypercubes.

Rank	Binary	Subset
0	0 0 0	{}
1	0 0 1	{3}
2	0 1 1	[2,3}
3	0 1 0	{2}
4	1 1 0	{1,2}
5	1 1 1	{1,2,3}
6	1 0 1	{1,3}
7	1 0 0	{1}

Sequential Subset Algorithm (direct ordering)

1. **for** i $\leftarrow$ 1 **to** n
2. **do** $a_i \leftarrow 0$
3. k $\leftarrow$ 0 // k is the cardinality

The above is for the first subset. The code below is for all later subsets.

1. i $\leftarrow$ 1
2. **while** $a_i = 1$
3. **do** i $\leftarrow$ i + 1
4. k $\leftarrow$ k - 1
5. $a_i \leftarrow 0$
6. $a_i \leftarrow 1$
7. k $\leftarrow$ k + 1
8. **if** k = n
9. **then exit**

How long does it take to generate the next subset?
Amortized time to calculate the next subset is:

$$\sum_{i=1}^{n} \frac{i}{2^i} = 2 - \frac{n+2}{2^n} = \Theta(1)$$

How to Parallelize Subset Generation?

We need to be able to unrank to a specified rank such that each process can start generating its share of the sequence. Unranking a value r is really just converting the number r into binary.

```
unrankSubset(r,n)
1. i ← 1
2. while (r> 0)
3. do a_i ← r mod 2
4.       r ← r/2 //integer division
5.       i ← i + 1
6. for j ← i to n
7. do a_j ← 0
```

Run-time: $T^*(n) = \Theta(\lg n)$

Parallel Generation of Subsets

Suppose we have p processes numbered $0 \ldots p-1$ and we want to generate 2^n subsets. Then the ith process generates $2^n/p$ subsets with the ranks: $i\frac{2^n}{p} \ldots (i+1)\frac{2^n}{p} - 1$

$[0 \ldots \frac{2^n}{p} - 1]$,	$[\frac{2^n}{p} \ldots 2\frac{2^n}{p} - 1]$,	$[2\frac{2^n}{p} \ldots 3\frac{2^n}{p} - 1]$	$\ldots$	$[\frac{p}{p-1}\frac{2^n}{p} \ldots 2^n - 1]$
P_0	P_1	P_2	$\ldots$	P_{p-1}

Sequential time: $T^*(n) = \Theta(2^n)$
Parallel time: $T_p(n) = \Theta(\frac{2^n}{p} + \lg n)$
Speedup: $S_p(n) = \Theta(\frac{2^n}{\frac{2^n}{p} + \lg n}) = \Theta(\frac{p}{1 + \frac{p \lg n}{2^n}})$

Parallel Generation of Subsets

Suppose we have p processes numbered $0 \ldots p-1$ and we want to generate 2^n subsets. Then the ith process generates subsets with the ranks:

$$i\frac{2^n}{p} \ldots (i+1)\frac{2^n}{p} - 1$$

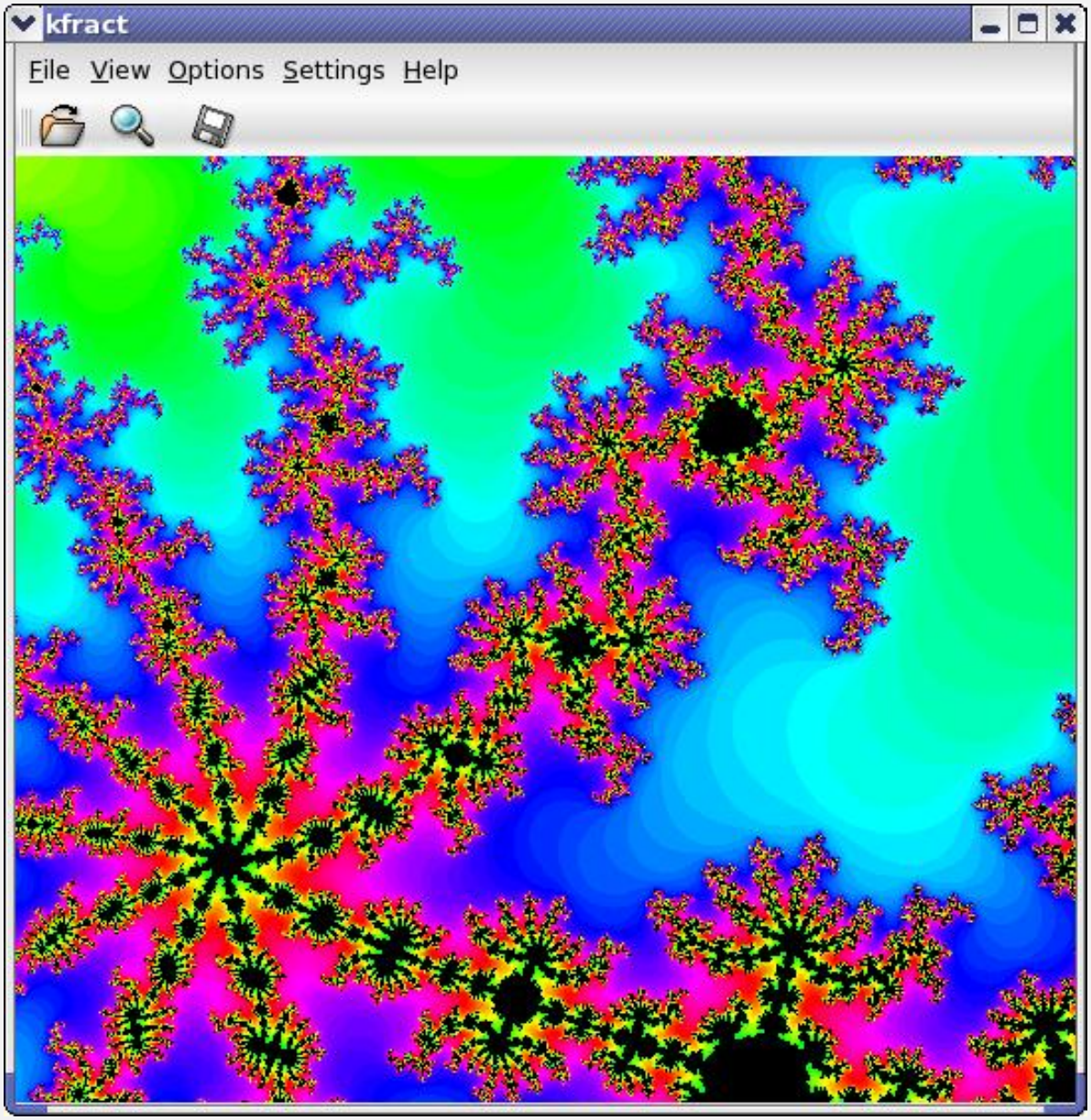
parallelSubsets(i, n, p)

1. a[1..n] $\leftarrow$ unrankSubset($i\frac{2^n}{p}$, n)
2. Generate the next $\frac{2^n}{p} - 1$ subsets
3. **if** (me == 0)
4. **for** k $\leftarrow$ 1 **to** p-1
5. **do** recv(&source, P_{ANY})
6. **else**
7. send(&me, P_0)

Parallel Generation of Other Combinatorial Objects

- To generate subsets in graycode order or lexicographic order requires an unrank function. This turns out to be more complex.
- For other combinatorial objects, again we need to come up with an unrank function. Each unrank function involves understanding the properties of the object.
- We have created a *CombAlgs* library that solves this problem. It was developed by undergraduate students Elizabeth Elzinga, Jeff Shellman and graduate student Brad Seewald.

Mandelbrot Set



Mandelbrot Set

- A Mandelbrot Set is a set of points on a complex plane that are quasi stable (that is they will increase and decrease, but not exceed some limit) when computed by iterating a function.
- A commonly used function for iterating over the points is:

$$\begin{aligned} z_{k+1} &= z_k^2 + c \\ z_0 &= 0 \end{aligned}$$

  where $z = z_{\text{real}} + iz_{\text{imag}}$ and $i = \sqrt{-1}$.
- The complex number c is the position of the point in the complex plane. The iterations are continued until the magnitude of z, defined as $|z| = \sqrt{z_{\text{real}}^2 + z_{\text{imag}}^2}$, is greater than 2 or the number of iterations reaches some arbitrary limit.
- The graphic display is based on the number of iterations it takes for each point to reach the limit. The complex plane of interest is typically: $[-2:2, -2:2]$.

Mandelbrot Set (contd.)

Expanding the function described on the previous frame, we have:

$$z^2 = (z_{\text{real}} + iz_{\text{imag}})^2 = z_{\text{real}}^2 - z_{\text{imag}}^2 + 2iz_{\text{real}}z_{\text{imag}}$$

Thus to compute the value of z for the next iteration, we have:

$$\boxed{\begin{array}{lcl} z_{\text{real}} & = & z_{\text{real}}^2 - z_{\text{imag}}^2 + c_{\text{real}} \\ z_{\text{imag}} & = & 2z_{\text{real}}z_{\text{imag}} + c_{\text{imag}} \end{array}}$$

- Let the complex plane of interest be `[rmin:rmax, imin:imax]`.
- Suppose the display is of size `disp_height` $\times$ `disp_width`. The each point (x, y) needs to be scaled as follows:

$$\texttt{c.real} = \texttt{rmin} + \texttt{x} * \overbrace{(\texttt{rmax} - \texttt{rmin})/\texttt{disp\_width}}^{\texttt{scale\_real}};$$

$$\texttt{c.imag} = \texttt{imin} + \texttt{y} * \overbrace{(\texttt{imax} - \texttt{imin})/\texttt{disp\_height}}^{\texttt{scale\_imag}};$$

Sequential Mandelbrot Set

Suppose we have a function `cal_pixel(c)`, that returns the number of iterations used starting from the point c in the complex plane.

```
for(x=0; x<disp_width; x++)
   for(y=0; y<disp_height; y++) {
      c.real = rmin + ((float) x * scale_real)
      c.imag = imin + ((float) y * scale_imag)
      color = cal_pixel(c)
      display(x,y,color)
   }
```

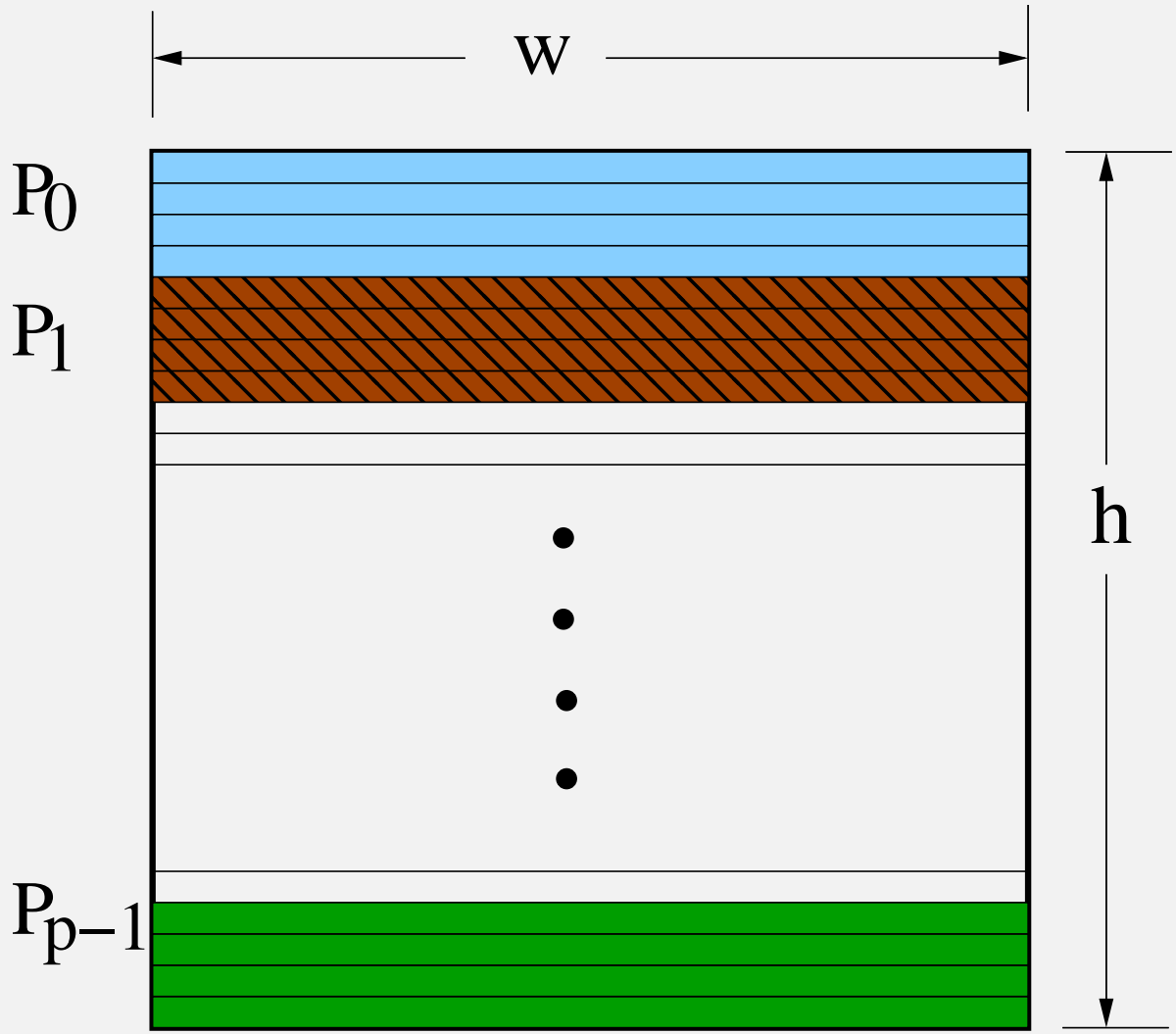
Let m be the maximum number of iterations in `cal_pixel()`.
Let $n = \text{disp\_width} \times \text{disp\_height}$ be the total number of points.
Then the sequential run time is $\boxed{T^*(m,n) = O(mn)}$.

Parallel Mandelbrot Set

- Making each task be computing one pixel would be too fine-grained and lead to a lot of communication. So we will count computing one row of the display as one task.
- Two ways of assigning tasks to solve the problem in parallel.
  - **Static task assignment**. Each process does a fixed part of the problem. Three different ways of assigning tasks statically:
    - **Divide by groups of rows** (or columns)
    - **Round robin by rows** (or columns)
    - **Checkerboard mapping**
  - **Dynamic task assignment**. A **work-pool** is maintained that worker processes go to get more work. Each process may end up doing different parts of the problem for different inputs or different runs on the same input.

The above techniques are very general and apply to most parallel programs.

Static Task Assignment: By Rows



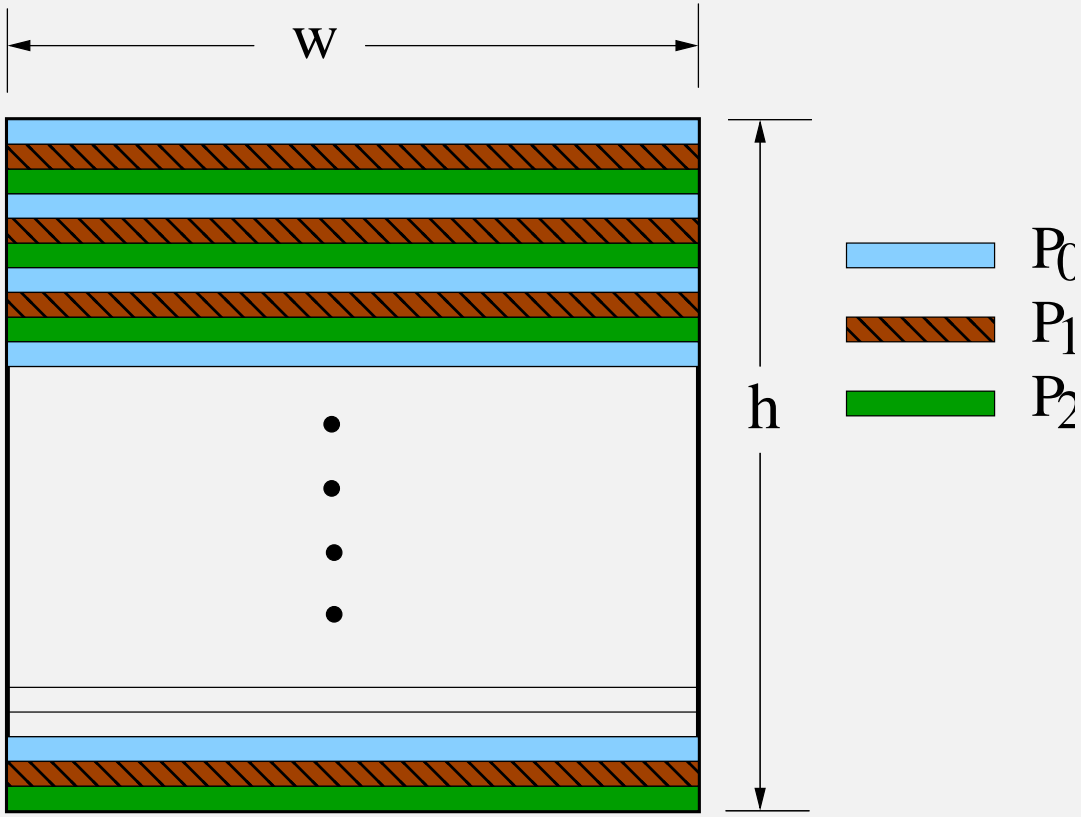
The display width = w, and the display height = h.
Each process handles at most $\lceil h/p \rceil$ rows.
Total number of points = $n = h \times w$.

Static Task Assignment: By Rows (Pseudo-code)

```
mandelbrot(h,w,p,id)
//w = display width, h = display height
//p+1 processes, process number id is 0 ≤ id ≤ p
//arrays c[0...w-1] and color[0...w-1]

  if (id == 0) { //Process 0 handles the display
    for (i=0; i<h; i++) receive one row at a time
      recv(c,color, P_any)
      display(c,color)
  } else { //the worker processes
    id = id - 1 //renumber id to 0..p-1 from 1..p
    share = ⌈h/p⌉
    start = id * share
    end = (id + 1) * share - 1
    if (id == p-1) end = h - 1 //boundary condition
    for (y=start; y<=end; y++) {
      for(x=0; x<w; x++) {
        c[x].real = rmin + ((float) x * scale_real)
        c[x].imag = imin + ((float) y * scale_imag)
        color[x] = cal_pixel(c[x])
      }
      send(c, color, P_0)
    }
  }
```

Static Task Assignment: Round Robin



Each process handles at most $\lceil h/p \rceil$ rows.
Total number of points $= n = h \times w$.

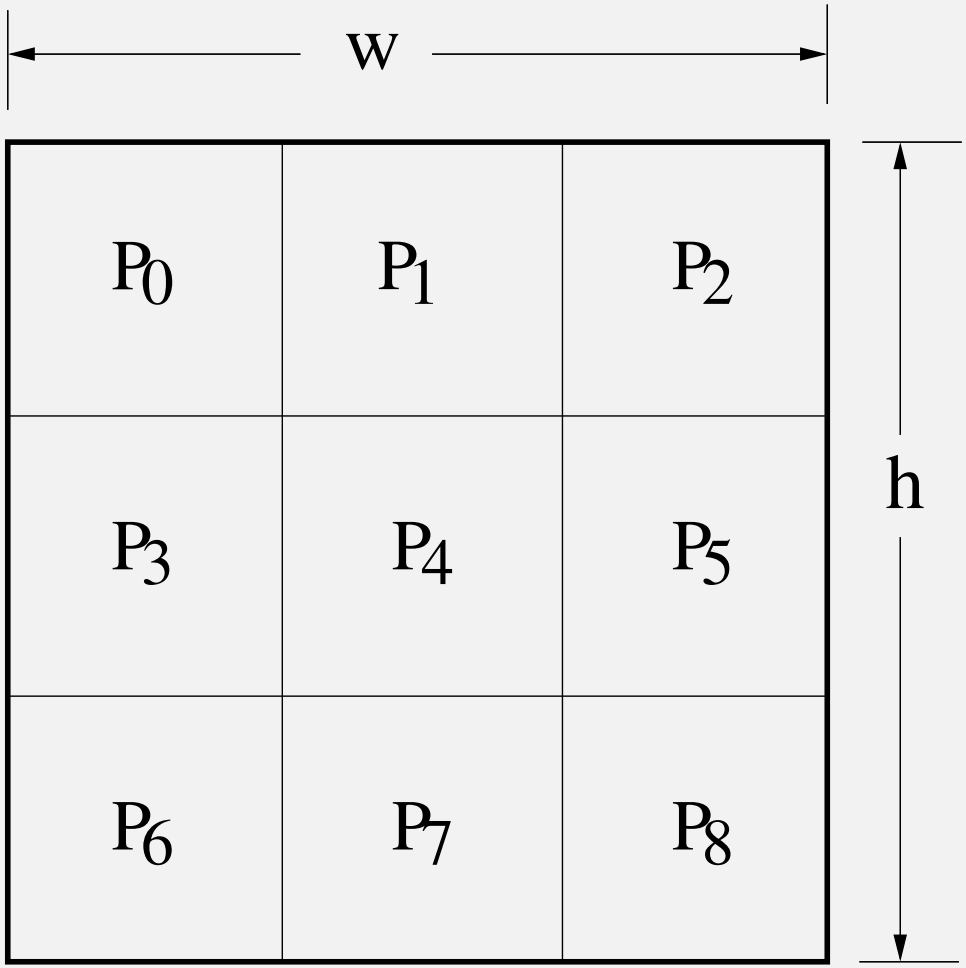
Static Task Assignment: Round Robin (Pseudo-code)

```
mandelbrot(h,w,p,id)
//w = display width, h = display height
//p+1 processes, id = process number, 0 ≤ id ≤ p
//arrays c[0...w-1] and color[0...w-1]

    if (id == 0) { //Process 0 handles the display
        for (i=0; i<h; i++) receive one row at a time
            recv(c,color, P_any)
            display(c,color)
    } else { //the worker processes
        for (y=id-1; y<h; y=y+p) {
            for(x=0; x<w; x++) {
                c[x].real = rmin + ((float) x * scale_real)
                c[x].imag = imin + ((float) y * scale_imag)
                color[x] = cal_pixel(c[x])
            }
            send(c, color, P_0)
        }
    }
```

→*simpler code than task assignment by rows*

Static Task Assignment: Checkerboard



Assume that $p = m \times m$ is a square number.
Each process handles at most $\lceil n/p \rceil$ points.

Static Task Assignment: Checkerboard (Pseudo-code)

Column-major overall and also within each square.

```
mandelbrot(h,w,p,id)
//w = display width, h = display height
//p = m × m = number of processes, id = process number, 0 ≤ id ≤ p − 1
//arrays c[0...⌈w/m⌉ − 1][0...⌈h/m⌉ − 1] and color[0...⌈w/m⌉ − 1][0...⌈h/m⌉ − 1]

  if (id == 0) { //Process 0 handles the display
    for (i=0; i<p; i++) receive one sub-square at a time
      recv(c,color, P_any)
      display(c,color)
  } else { //the worker processes
    id = id - 1 //renumber id to 0..p-1 from 1..p
    m = √p; r = ⌊id/m⌋; c = id % m;
    my_h = ⌈h/m⌉; my_w = ⌈w/m⌉
    startx = c * my_w; starty = r * my_h
    endx = startx + my_w; if (c == m-1) endx = w
    endy = starty + my_h; if (r == m-1) endy = h
    for (y=starty; y<endy; y++) {
      for(x=startx; x<endx; x++) {
        c[x][y].real = rmin + ((float) x * scale_real)
        c[x][y].imag = imin + ((float) y * scale_imag)
        color[x][y] = cal_pixel(c[x][y])
      }
    }
    send(c, color, P_0)
  }
```

Static Task Assignment: Checkerboard version 2

Row-major overall and also within each square.

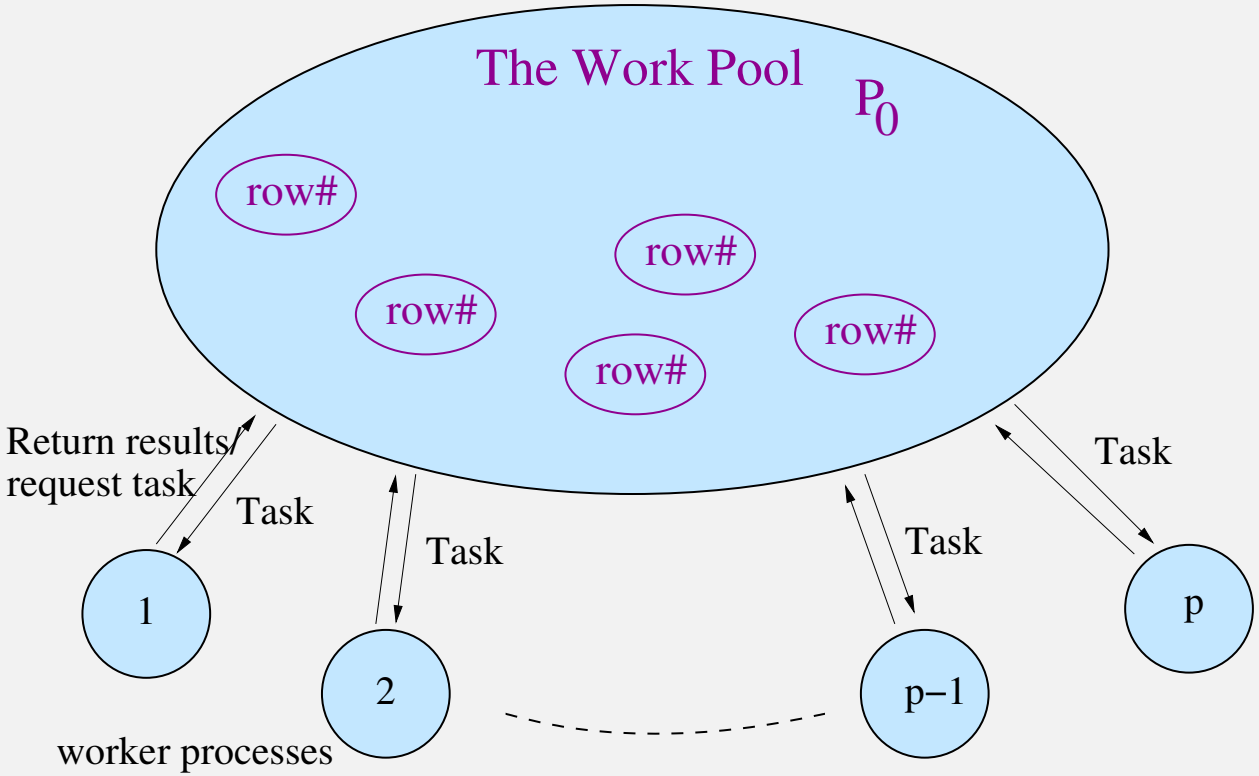
```
mandelbrot(h,w,p,id)
//w = display width, h = display height
//p = m × m = number of processes, id = process number, 0 ≤ id ≤ p − 1
//arrays c[0...⌈w/m⌉ − 1][0...⌈h/m⌉ − 1] and color[0...⌈w/m⌉ − 1][0...⌈h/m⌉ − 1]

    if (id == 0) { //Process 0 handles the display
        for (i=0; i<p; i++) receive one sub-square at a time
            recv(c,color, P_any)
            display(c,color)
    } else { //the worker processes
        id = id - 1 //renumber id to 0..p-1 from 1..p
        m = √p; r = ⌊id/m⌋; c = id % m;
        my_h = ⌈h/m⌉; my_w = ⌈w/m⌉
        startx = r * my_h; starty = c * my_w
        endx = startx + my_h; if (c == m-1) endx = h
        endy = starty + my_w; if (r == m-1) endy = w
        for (x=startx; x<endx; x++) {
            for(y=starty; y<endy; y++) {
                c[x][y].real = rmin + ((float) x * scale_real)
                c[x][y].imag = imin + ((float) y * scale_imag)
                color[x][y] = cal_pixel(c[x][y])
            }
        }
        send(c, color, P_0)
    }
```

Comparison of the three Static Task Assignments

- In the computation of the Mandelbrot set, some regions take more iterations and others take fewer iterations in an unpredictable fashion. With any of three task assignments some processes may have a bigger load while others are idling. This is an instance of the more general **load balancing** problem.
- The round robin approach is likely to be the least unbalanced of the three static task assignments.
- We can do better if we use a dynamic load balancing method: the **work-pool** approach.

Dynamic Task Assignment: The Work Pool



The work pool holds a collection of tasks to be performed. Processes are supplied with tasks as soon as they finish previously assigned task. In more complex work pool problems, processes may even generate new tasks to be added to the work pool.

- *task* (for Mandelbrot set): one row to be calculated
- *coordinator process (process 0)*: holds the work pool, which simply consists of number of rows that still need to be done

The Work Pool Pseudo-Code

```
mandelbrot(h, w, p, id)
// p+1 processes, numbered id=0,...,p.

    if (id == 0) {
        count = 0
        row = 0
        for(k=1; k<=p; k++) {
            send(&row, P_k, data)
            count++; row++;
        }
        do {
            recv(&id, &r, color, P_any, result)
            count-
            if (row < h) {
                send(&row, P_id, data)
                row++; count++
            } else {
                send(&row, P_id, termination)
            }
            display(r, color)
        } while (count > 0)
```

The Work Pool Pseudo-Code (contd.)

```
} else {   // the worker processes
    recv(&y, P0, ANYTAG, &source_tag)
    while (source_tag == data) {
        c.imag = imin + y * scale_imag
        for (x=0; x<w; x++) {
            c.real = rmin + x * scale_real
            color[x] = cal_pixel(c)
        }
        send(&id, &y, color, P0, result)
        recv(&y, P0, ANYTAG, &source_tag)
    }
}
```

Analysis of the Work-Pool Approach

- Let m be the maximum number of iterations in `cal_pixel()` function. Then sequential time is $\boxed{T^*(m,n) = O(mn)}$.
- **Phase I**. $T_{\text{comm}}(n) = p(t_{\text{startup}} + t_{\text{data}})$.
- **Phase II**. Additional $h - p$ values are sent to the worker processes. $T_{\text{comm}}(n) = (h-p)(t_{\text{startup}} + t_{\text{data}})$. Each worker process computes $\leq n/p$ points. Thus we have $T_{\text{comm}}(n) \leq (m \times n)/p = O(mn/p)$.
- **Phase III**. A total of h rows are sent back, each w elements wide.

$$\begin{aligned} T_{\text{comm}}(n) &= h(t_{\text{startup}} + wt_{\text{data}}) \\ &= ht_{\text{startup}} + hwt_{\text{data}} = O(ht_{\text{startup}} + nt_{\text{data}}) \end{aligned}$$

- The overall parallel run time:

$$\boxed{T_p(n) = O(ht_{\text{startup}} + (n+h)t_{\text{data}} + \tfrac{mn}{p})}$$

- The speedup is:

$$\boxed{S_p(n) = O\left(\frac{p}{1 + \frac{hp}{mn}t_{\text{startup}} + \frac{(n+h)p}{mn}t_{\text{data}}}\right)}$$

Further Examples of Embarrassingly Parallel Problems

- Serving static files on a webserver.
- Event simulation and reconstruction in particle physics.
- BLAST searches in bioinformatics for multiple queries.
- Large scale face recognition that involves comparing thousands of arbitrary acquired faces with similarly large number of previously stored faces.
- Computer simulations comparing many independent scenarios, such as climate models, crystal growth and many others.
- Rendering of computer graphics. In ray tracing, each pixel may be rendered independently. In computer animation, each frame may be rendered independently.
- Distributed relational database queries using distributed set processing
- MapReduce programs on massive data sets.