Outline: Embarrassingly Parallel Problems

- what they are
- Mandelbrot Set computation
  - cost considerations
  - static parallelization
  - dynamic parallelizations and its analysis
- Monte Carlo Methods
- parallel random number generation

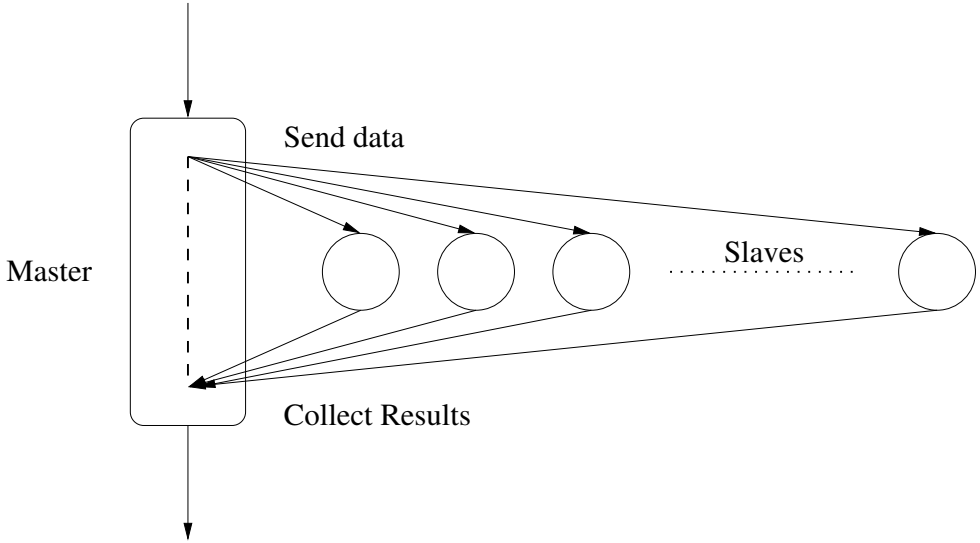
Ref: Lin and Snyder Ch 5, Wilkinson and Allen Ch 3

Admin: reminder - pracs this week, get your NCI accounts!; parallel programming poll

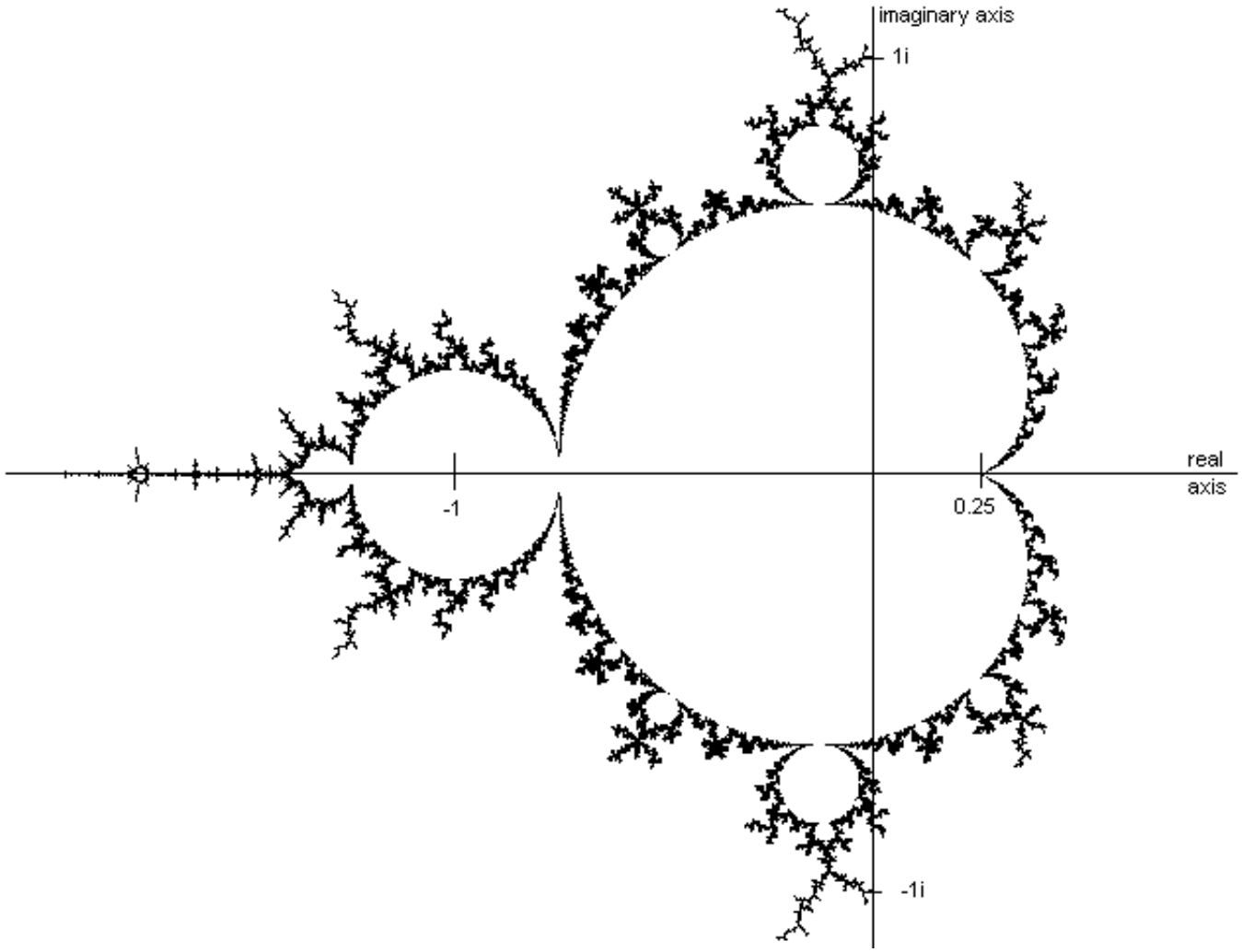
News: Improving Energy Efficiency and Exploiting Parallelism with Processing in Memory and Near-Data Processing

Embarrassingly Parallel Problems

- computation can be divided into completely independent parts for execution by separate processors (correspond to totally disconnected computational graphs)
  - infrastructure: Blocks of Independent Computations (BOINC) project
  - SETI@home and Folding@Home are projects solving very large such problems
- part of an application may be embarrassingly parallel
- distribution and collection of data are the key issues (can be non-trivial and/or costly)
- frequently uses the master/slave approach ($p-1$ speedup)



Example#1: Computation of the Mandelbrot Set



The Mandelbrot Set

- set of points in complex plane that are quasi-stable
- computed by iterating the function

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

  - z and c are complex numbers ($z = a + bi$)
  - z initially zero
  - c gives the position of the point in the complex plane
- iterations continue until $|z| > 2$ or some arbitrary iteration limit is reached

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

- enclosed by a circle centered at (0,0) of radius 2

Evaluating 1 Point

```
typedef struct complex{float real, imag;} complex;
const int MaxIter = 256;

int calc_pixel(complex c){
  int count = 0;
  complex z = {0.0, 0.0};
  float temp, lengthsq;
  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 < MaxIter);
  return count;
}
```

Building the Full Image

Define:

- min. and max. values for **c** (usually -2 to 2)
- number of horizontal and vertical pixels

```
for (x = 0; x < width; x++)
  for (y = 0; y < height; y++){
    c.real = min.real + ((float) x * (max.real - min.real) / width);
    c.imag = min.imag + ((float) y * (max.imag - min.imag) / height)
    color = calc_pixel(c);
    display(x, y, color);
}
```

Summary:

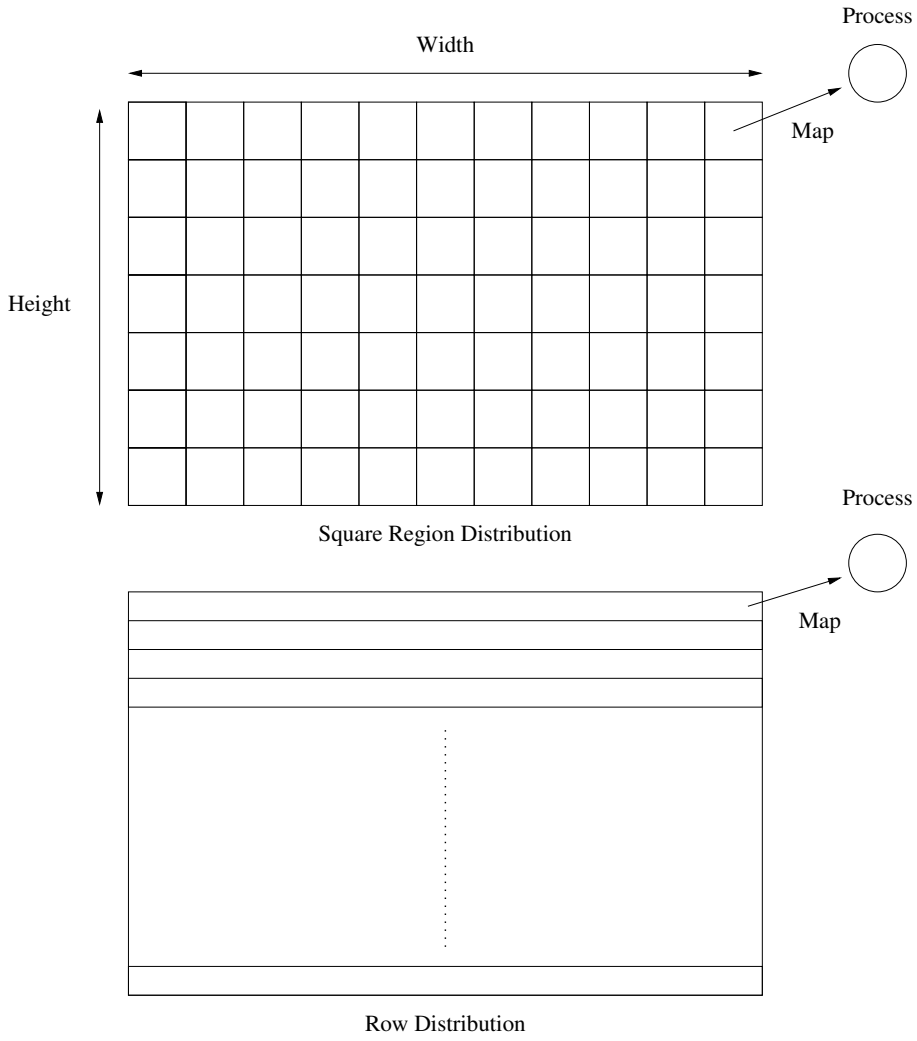
- **width** $\times$ **height** totally independent tasks
- each task can be of different length

Cost Considerations on NCI's Raijin

- 10 flops per iteration
- maximum 256 iterations per point
- approximate time on one Raijin core:

  $10 \times 256/(8 \times 2.6 \times 10^9) \approx 0.12\text{usec}$
- between two nodes the time to communicate single point to slave and receive result $\approx 2 \times 2\text{usec}$ (latency limited)
- conclusion: cannot parallelize over individual points
- also must allow time for master to send to all slaves before it can return to any given process

Parallelisation: Static



Static Implementation

Master:

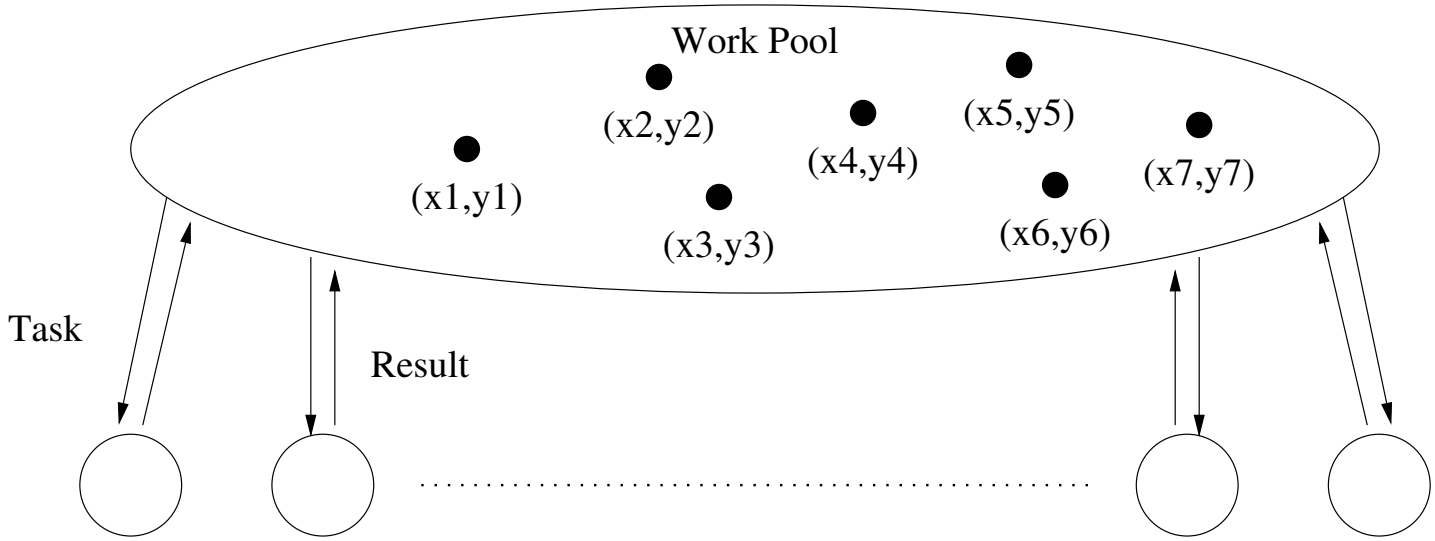
```
for (slave = 1, row = 0; slave < nproc; slave++) {
  send(&row, slave);
  row = row + height/nproc;
}
for (npixel = 0; npixel < (width * height); npixel++) {
  recv(&x, &y, &color, any_processor);
  display(x, y, color);
}
```

Slave:

```
const int master = 0; // proc. id
recv(&firstrow, master);
lastrow = MIN(firstrow + height/nproc, height);
for (x = 0; x < width; x++)
  for (y = firstrow; y < lastrow; y++) {
    c.real = min.real + ((float) x * (max.real - min.real)/width);
    c.imag = min.imag + ((float) y * (max.imag - min.imag)/height);
    color = calc_pixel(c);
    send(&x, &y, &color, master);
  }
```

Dynamic Task Assignment

- discussion point: why would we expect static assignment to be sub-optimal for the Mandelbrot set calculation? Would any regular static decomposition be significantly better (or worse)?
- pool of over-decomposed) tasks that are dynamically assigned to next requesting process



Processor Farm: Master

```
count = 0;
row = 0;
for (slave = 1; slave < nproc; k++){
  send(&row, slave, data_tag);
  count++;
  row++;
}
do {
  recv(&slave, &r, &color, any_proc, result_tag);
  count--;
  if (row < height) {
     send(&row, slave, data_tag);
     row++;
     count++;
  } else
    send(&row, slave, terminator_tag);
  display_vector(r, color);
} while (count > 0);
```

Processor Farm: Slave

```
const int master = 0; //proc id.
recv(&y, master, any_tag, source_tag);
while (source_tag == data_tag) {
  c.imag = min.imag + ((float) y * (max.imag - min.imag)/height);
  for (x = 0; x < width; x++) {
    c.real = min.real + ((float) x * (max.real - min.real)/width);
    color[x] = calc_pixel(c);
  }
  send(&myid, &y, color, master, result_tag);
  recv(&y, master, source_tag);
}
```

Analysis

Let p, m, n, I denote `nproc, height, width, MaxIter`:

- sequential time: (t_f denotes floating point operation time)

  $t_{\text{seq}} \leq I \cdot mn \cdot t_f = O(mn)$

- parallel communication 1: (neglect t_h term, assume message length of 1 word)

  $t_{\text{com1}} = 2(p-1)(t_s + t_w)$

- parallel computation:

  $t_{\text{comp}} \leq \frac{I \cdot mn}{p-1} t_f$

- parallel communication 2:

  $t_{\text{com2}} = \frac{m}{p-1}(t_s + t_w)$

- overall:

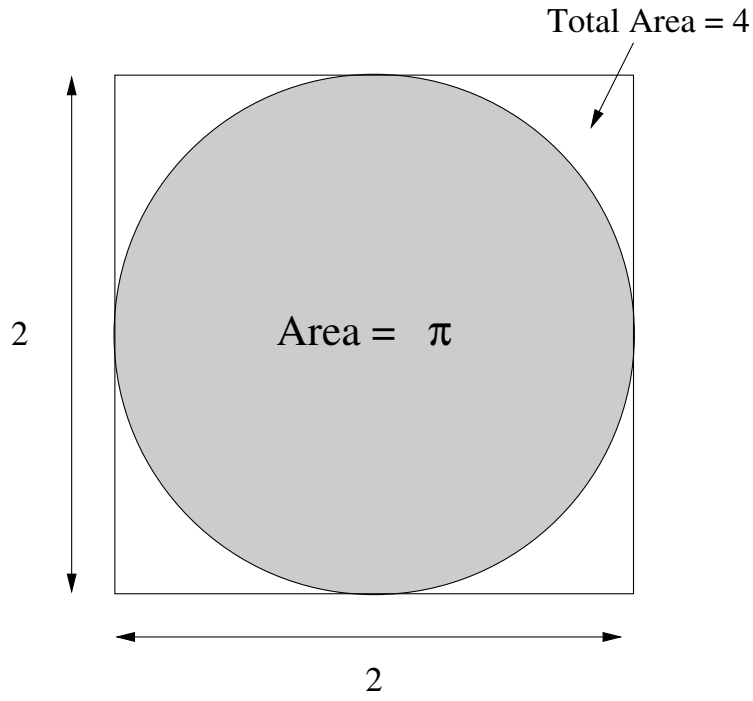
  $t_{\text{par}} \leq \frac{I \cdot mn}{p-1} t_f + (p - 1 + \frac{m}{p-1})(t_s + t_w)$

Discussion point: What assumptions have we been making here? Are there any situations where we might still have poor performance, and how could we mitigate this?

Example#2: Monte Carlo Methods

- use random numbers to solve numerical/physical problems
- evaluation of π by determining if random points in the unit square fall inside a circle

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



Monte Carlo Integration

- evaluation of integral ($x_1 \leq x_i \leq x_2$)

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

- example

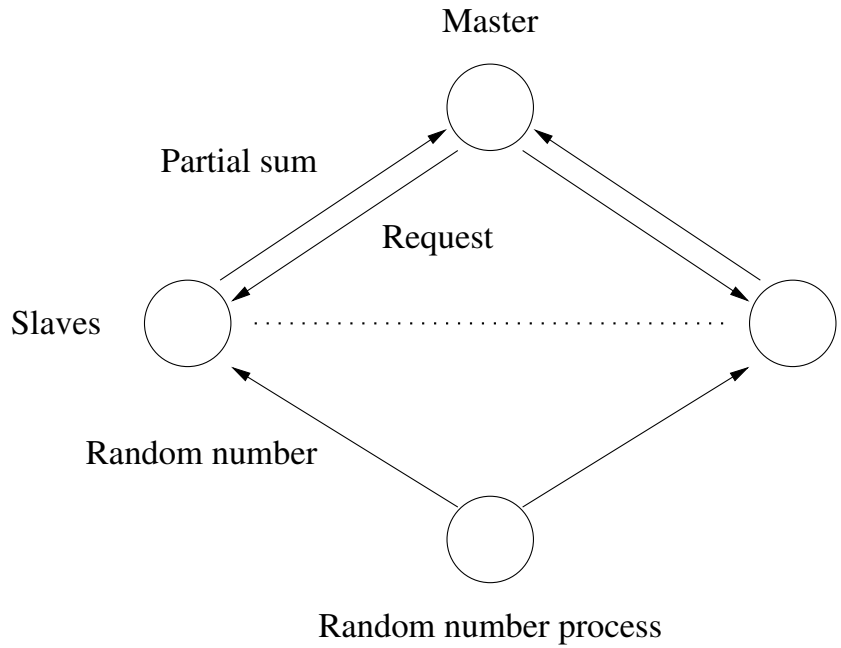
$$I = \int_{x_1}^{x_2} (x^2 - 3x)\,dx$$

```
sum = 0;
for (i = 0; i < N; i++) {
    xr = rand_v(x1, x2);
    sum += xr * xr - 3 * xr;
}
area = sum * (x2 - x1) / N;
```

- where `rand_v(x1, x2)` computes a pseudo-random number between `x1` and `x2`

Parallelization

- only problem is ensuring each process uses a different random number and that there is no correlation
- one solution is to have a unique process (maybe the master) issuing random numbers to the slaves



Parallel Code: Integration

Master (process 0):

```
for (i = 0; i < N/n; i++) {
  for (j = 0; j < n; j++)
    xr[j] = rand_v(x1, x2);
  recv(any_proc,req_tag,&p_src);
  send(xr, n, p_src, comp_tag);
}
for (i=1; i < nproc; i++) {
   recv(i, req_tag);
   send(i, stop_tag);
}
sum = 0;
reduce_add(&sum, p_group);
```

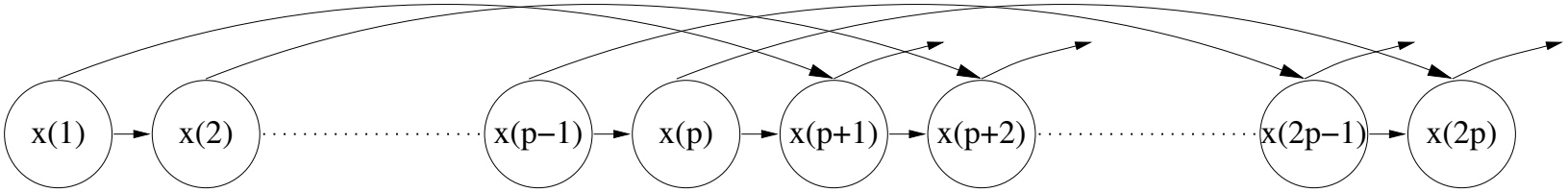
Slave:

```
const int master = 0; //proc id.
sum = 0;
send(master, req_tag);
recv(xr, &n, master, tag);
while (tag == comp_tag) {
  for (i = 0; i < n; i++)
    sum += xr[i]*xr[i] - 3*xr[i];
  send(0, req_tag);
  recv(xr, n, master, &tag);
}
reduce_add(&sum, P_group);
```

Question: performance problems with this code?

Parallel Random Numbers

- linear congruential generators $x_{i+1} = (ax_i + c) \bmod m$ (a, c, and m are constants)
- using property $x_{i+p} = (A(a,p,m)x_i + C(c,a,p,m)) \bmod m$, we can generate the first p random numbers sequentially to repeatedly calculate the next p numbers in parallel



Summary: embarrassingly parallel problems

- defining characteristic: tasks do not need to communicate
- non-trivial however: providing input data to tasks, assembling results, load balancing, scheduling, heterogeneous compute resources, costing
  - static task assignment (lower communication costs) vs.
    dynamic task assignment + overdecomposition (better load balance)
- Monte Carlo or ensemble simulations are a big use of computational power!
- the field of grid computing arose to solve this issue