


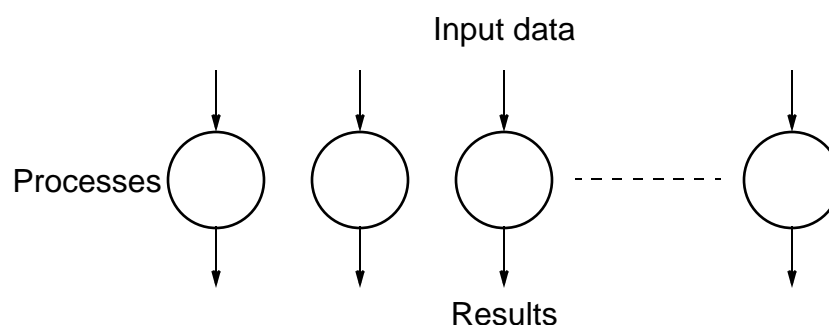
## Parallel Techniques

- Embarrassingly Parallel Computations
- Partitioning and Divide-and-Conquer Strategies
- Pipelined Computations
- Synchronous Computations
- Asynchronous Computations  Not covered in 1st edition of textbook
- Load Balancing and Termination Detection

## Embarrassingly Parallel Computations

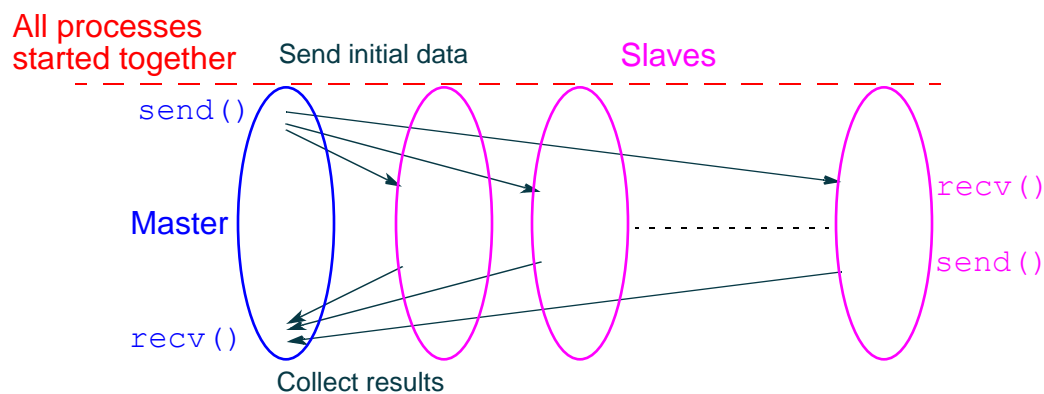
## Embarrassingly Parallel Computations

A computation that can **obviously** be divided into a number of completely independent parts, each of which can be executed by a separate process(or).



No communication or very little communication between processes  
Each process can do its tasks without any interaction with other processes

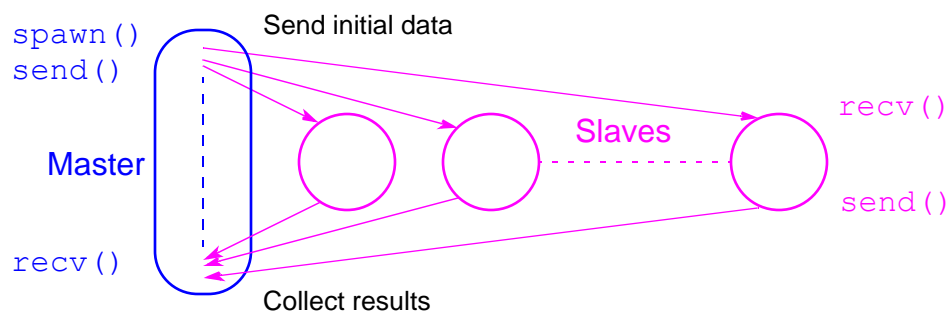
## Practical embarrassingly parallel computation with static process creation and master-slave approach



MPI approach

## Practical embarrassingly parallel computation with dynamic process creation and master-slave approach

Start Master initially



PVM approach

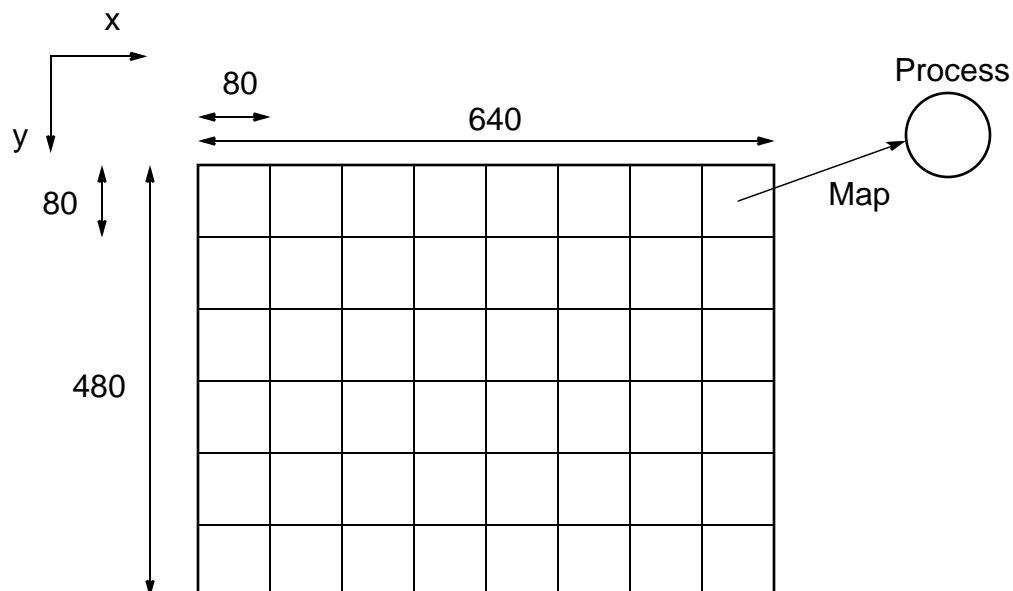
## Embarrassingly Parallel Computation Examples

- Low level image processing
- Mandelbrot set
- Monte Carlo Calculations

## Low level image processing

Many low level image processing operations only involve local data with very limited if any communication between areas of interest.

## Partitioning into regions for individual processes.



Square region for each process (can also use strips)

## Some geometrical operations

### Shifting

Object shifted by  $x$  in the  $x$ -dimension and  $y$  in the  $y$ -dimension:

$$X = x + \Delta x$$

$$Y = y + \Delta y$$

where  $x$  and  $y$  are the original and  $X$  and  $Y$  are the new coordinates.

### Scaling

Object scaled by a factor  $S_x$  in  $x$ -direction and  $S_y$  in  $y$ -direction:

$$X = xS_x$$

$$Y = yS_y$$

### Rotation

Object rotated through an angle  $\theta$  about the origin of the coordinate system:

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

$$Y = -x \sin \theta + y \cos \theta$$

## Mandelbrot Set

Set of points in a 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  $z_{k+1}$  is the  $(k + 1)$ th iteration of the complex number  $z = a + bi$  and  $c$  is a complex number giving position of point in the complex plane. The initial value for  $z$  is zero.

Iterations continued until magnitude of  $z$  is greater than 2 or number of iterations reaches arbitrary limit. Magnitude of  $z$  is the length of the vector given by

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

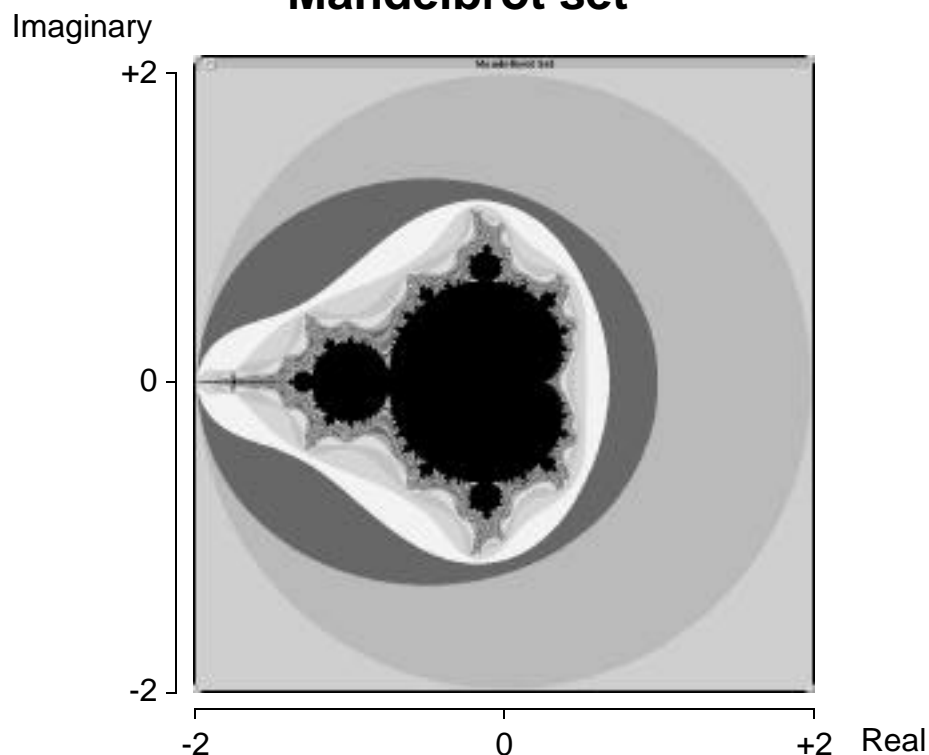
## Sequential routine computing value of one point returning number of iterations

```

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;                                /* 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;
}

```

## Mandelbrot set



# Parallelizing Mandelbrot Set Computation

## Static Task Assignment

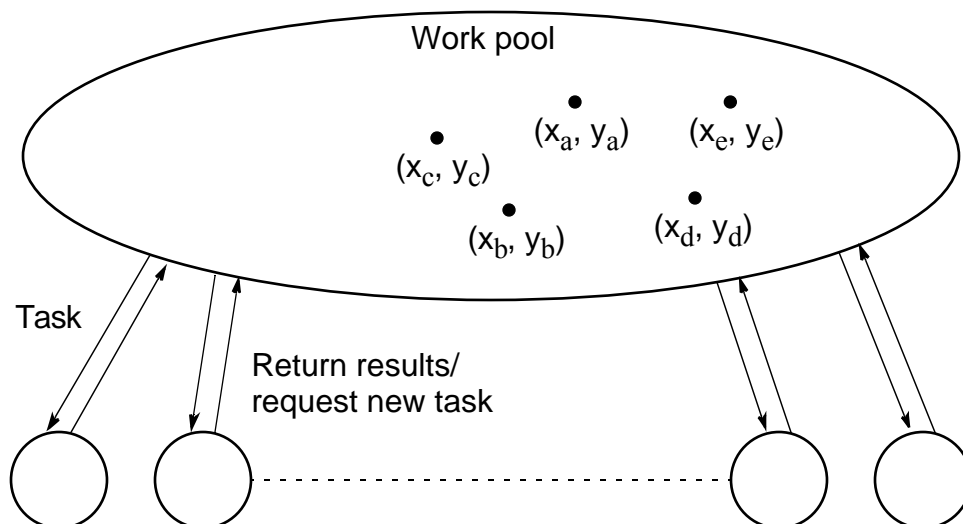
Simply divide the region in to fixed number of parts, each computed by a separate processor.

Not very successful because different regions require different numbers of iterations and time.

## Dynamic Task Assignment

Have processor request regions after computing previous regions

## Dynamic Task Assignment Work Pool/Processor Farms



## Monte Carlo Methods

Another embarrassingly parallel computation.

Monte Carlo methods use of random selections.

### Example - To calculate

Circle formed within a square, with unit radius so that square has sides  $2 \times 2$ . Ratio of the area of the circle to the square given by

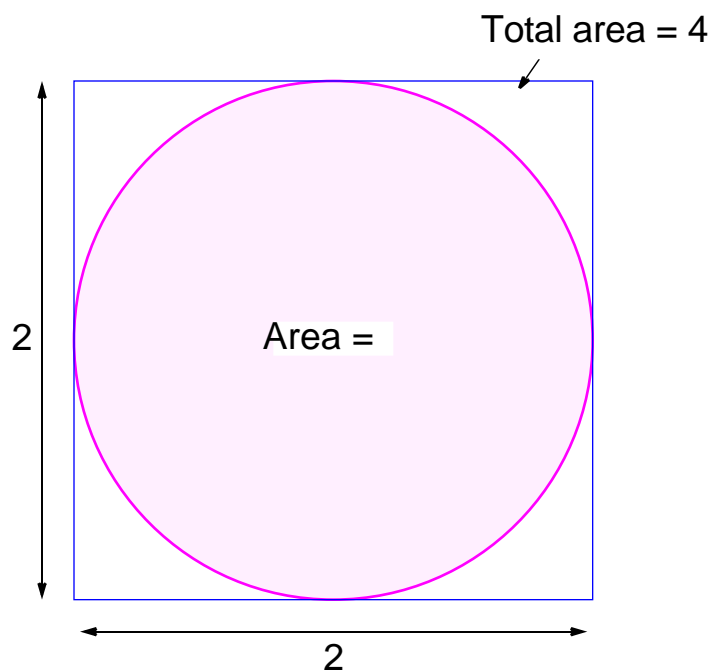
$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{(1)^2}{2 \times 2} = \frac{1}{4}$$

Points within square chosen randomly.

Score kept of how many points happen to lie within circle.

Fraction of points within the circle will be  $\pi/4$ , given a sufficient number of randomly selected samples.





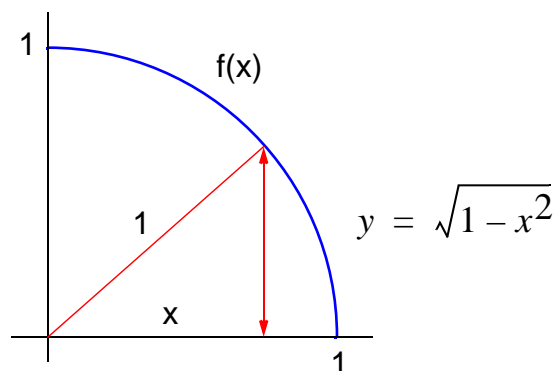
## Computing an Integral

One quadrant of the construction can be described by integral

$$\int_0^1 \sqrt{1-x^2} dx = \frac{\pi}{4}$$

Random pairs of numbers,  $(x_r, y_r)$  generated, each between 0 and 1.

Counted as in circle if  $y_r \leq \sqrt{1-x_r^2}$ ; that is,  $y_r^2 + x_r^2 \leq 1$ .



## Alternative (better) Method

Use random values of  $x$  to compute  $f(x)$  and sum values of  $f(x)$ :

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

where  $x_i$  are randomly generated values of  $x$  between  $x_1$  and  $x_2$ .

Monte Carlo method very useful if the function cannot be integrated numerically (maybe having a large number of variables)

## Example

Computing the integral

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

### Sequential Code

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

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

*For parallelizing Monte Carlo code, must address best way to generate random numbers in parallel - see textbook*

# Partitioning and Divide-and-Conquer Strategies

## Partitioning

Partitioning simply divides the problem into parts.

## Divide and Conquer

Characterized by dividing problem into subproblems of same form as larger problem. Further divisions into still smaller sub-problems, usually done by recursion.

Recursive divide and conquer amenable to parallelization because separate processes can be used for divided parts.  
Also usually data is naturally localized.

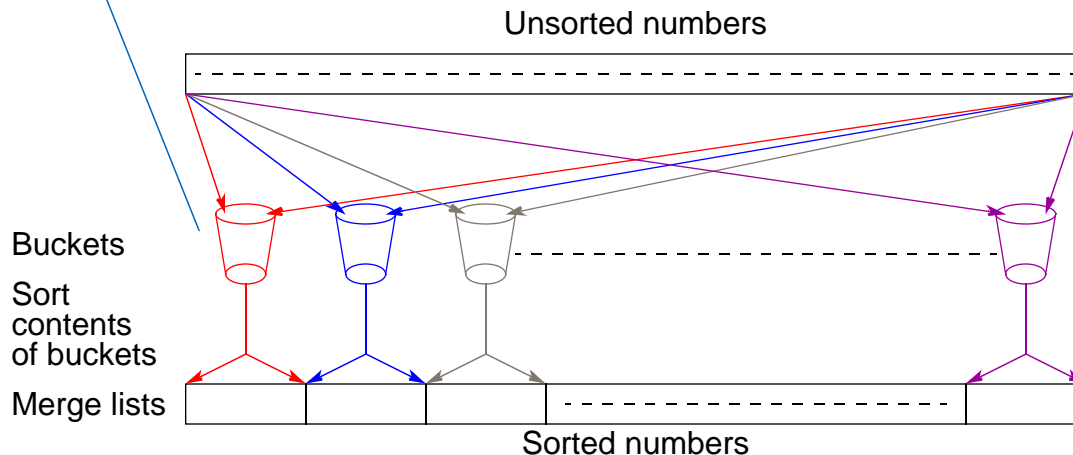
## Partitioning/Divide and Conquer Examples

Many possibilities.

- Operations on sequences of number such as simply adding them together
- Several sorting algorithms can often be partitioned or constructed in a recursive fashion
- Numerical integration
- $N$ -body problem

## Bucket sort

One "bucket" assigned to hold numbers that fall within each region.  
Numbers in each bucket sorted using a sequential sorting algorithm.



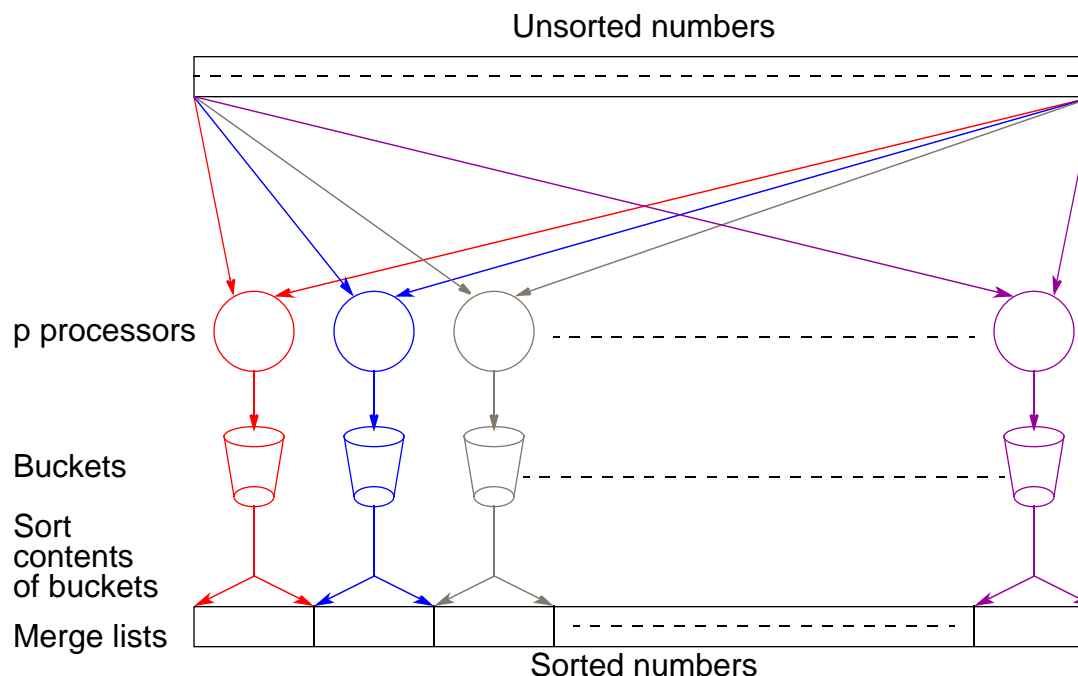
Sequential sorting time complexity:  $O(n \log(n/m))$ .

Works well if the original numbers uniformly distributed across a known interval, say 0 to  $a - 1$ .

## Parallel version of bucket sort

### Simple approach

Assign one processor for each bucket.



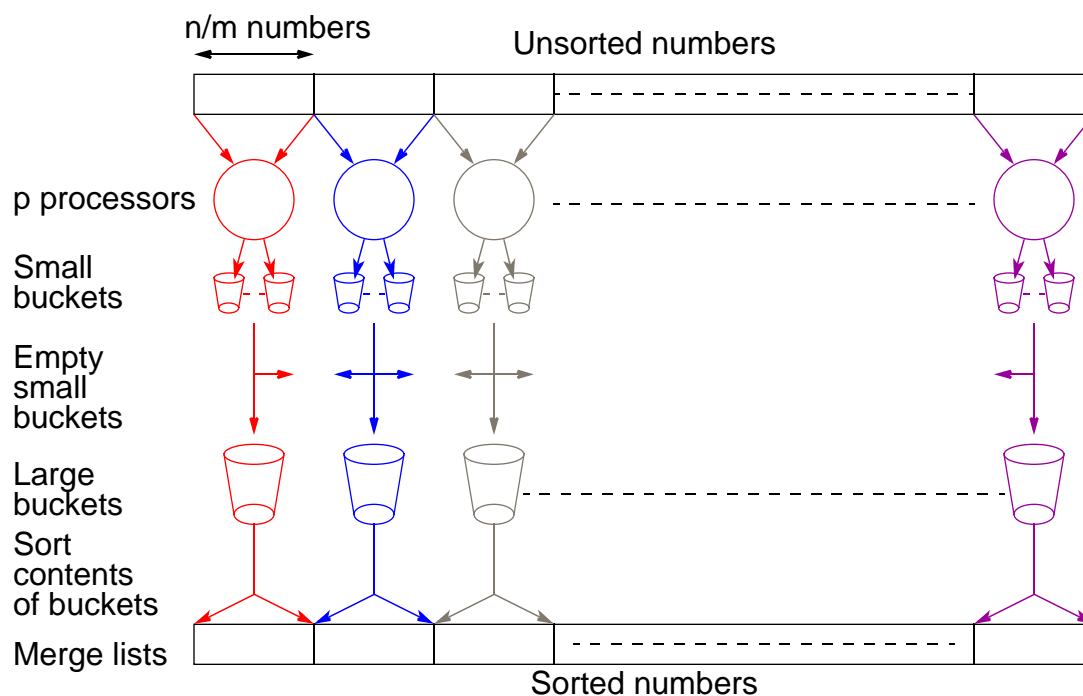
## Further Parallelization

Partition sequence into  $m$  regions, one region for each processor.

Each processor maintains  $p$  “small” buckets and separates the numbers in its region into its own small buckets.

Small buckets then emptied into  $p$  final buckets for sorting, which requires each processor to send one small bucket to each of the other processors (bucket  $i$  to processor  $i$ ).

## Another parallel version of bucket sort

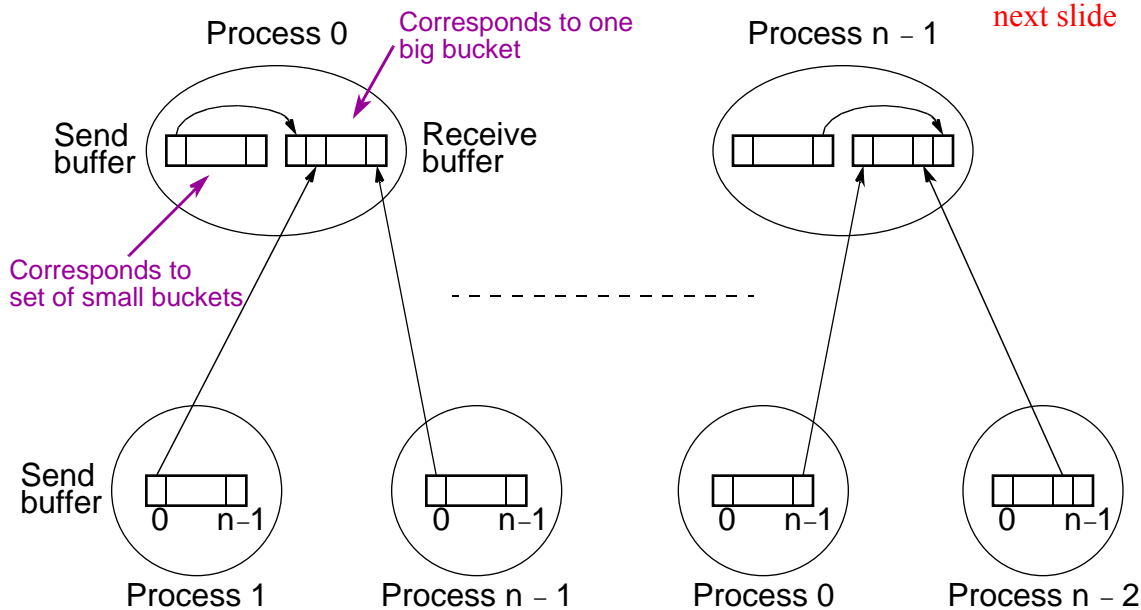


**Introduces new message-passing operation - all-to-all broadcast.**

## “all-to-all” broadcast routine

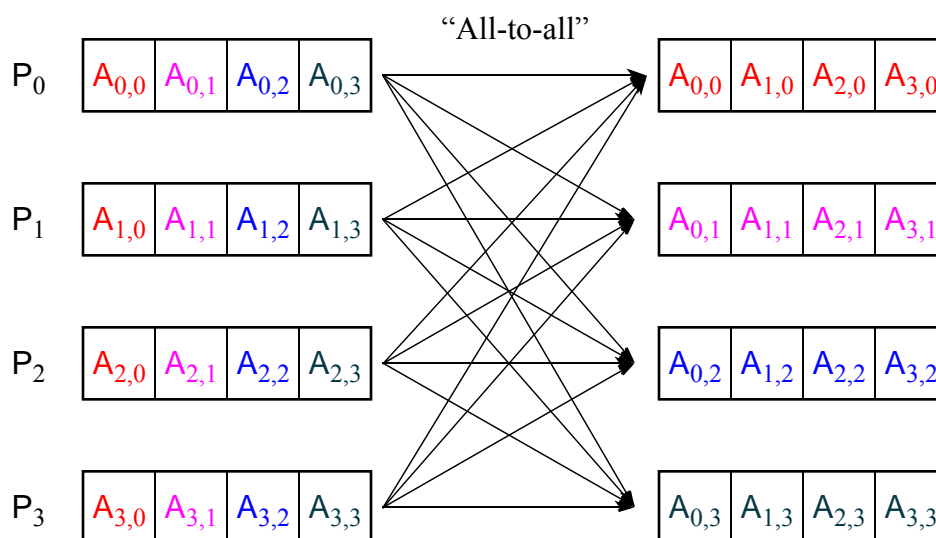
Sends data from each process to every other process

See also  
next slide



“all-to-all” routine actually transfers rows of an array to columns:

Tranposes a matrix.

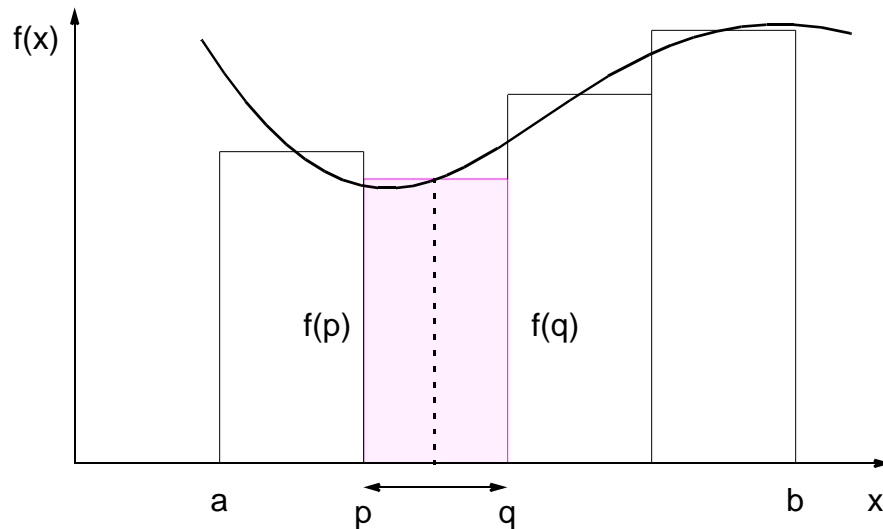


Effect of “all-to-all” on an array

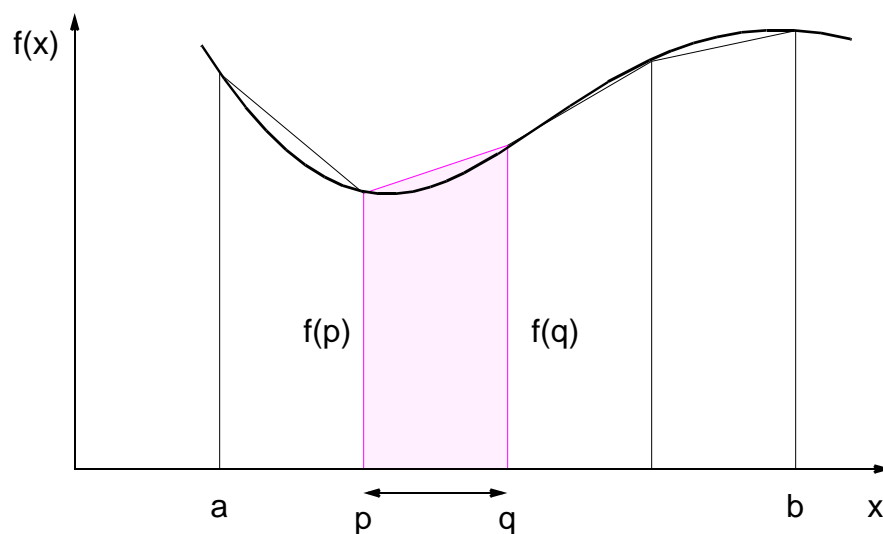
## Numerical integration using rectangles.

Each region calculated using an approximation given by rectangles:

Aligning the rectangles:



## Numerical integration using trapezoidal method

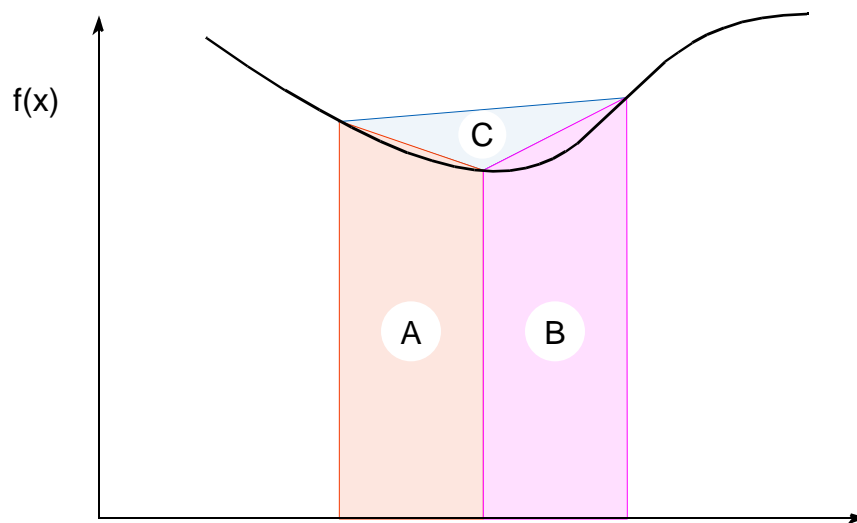


May not be better!



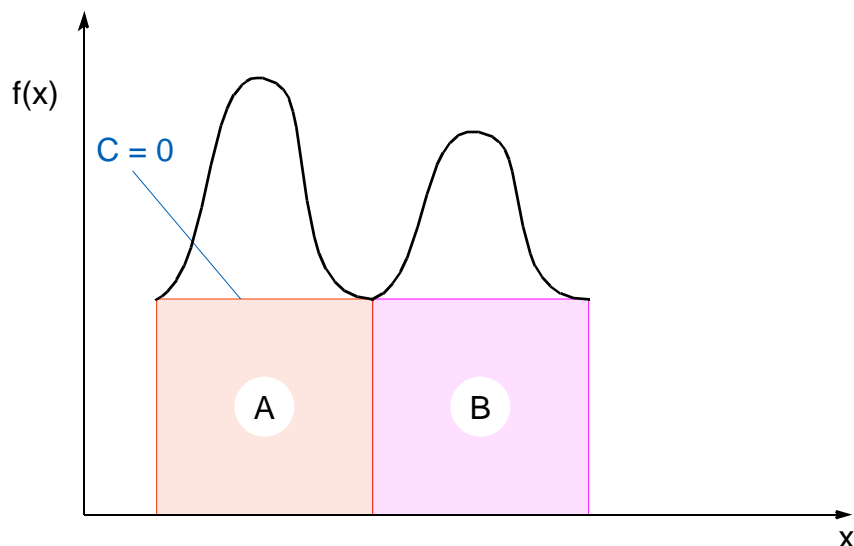
## Adaptive Quadrature

Solution adapts to shape of curve. Use three areas,  $A$ ,  $B$ , and  $C$ . Computation terminated when largest of  $A$  and  $B$  sufficiently close to sum of remain two areas .



## Adaptive quadrature with false termination.

Some care might be needed in choosing when to terminate.



Might cause us to terminate early, as two large regions are the same (i.e.,  $C = 0$ ).

## Simple program to compute

### Using C++ MPI routines

```

/*****
pi_calc.cpp calculates value of pi and compares with actual value (to 25
digits) of pi to give error.  Integrates function  $f(x)=4/(1+x^2)$ .
July 6, 2001 K. Spry CSCI3145
*****/
#include <math.h>           //include files
#include <iostream.h>
#include "mpi.h"
void printit();           //function prototypes
int main(int argc, char *argv[])
{
double actual_pi = 3.141592653589793238462643;  //for comparison later
int n, rank, num_proc, i;
double temp_pi, calc_pi, int_size, part_sum, x;
char response = 'y', respl = 'y';
MPI::Init(argc, argv); //initiate MPI
num_proc = MPI::COMM_WORLD.Get_size();
rank = MPI::COMM_WORLD.Get_rank();
if (rank == 0) printit();           /* I am root node, print out welcome */
while (response == 'y') {
    if (respl == 'y') {
        if (rank == 0) {           /*I am root node*/
            cout << "_____ " << endl;
            cout << "\nEnter the number of intervals: (0 will exit)" << endl;
            cin >> n;
        }
    }
    else n = 0;
}

```

```

MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0);    //broadcast n
if (n==0) break;                             //check for quit condition
else {
    int_size = 1.0 / (double) n; //calcs interval size
    part_sum = 0.0;
    for (i = rank + 1; i <= n; i += num_proc) { //calcs partial sums
        x = int_size * ((double)i - 0.5);
        part_sum += (4.0 / (1.0 + x*x));
    }
    temp_pi = int_size * part_sum;
    //collects all partial sums computes pi
MPI::COMM_WORLD.Reduce(&temp_pi,&calc_pi, 1, MPI::DOUBLE, MPI::SUM, 0);

```

```

    if (rank == 0) { /*I am server*/
        cout << "pi is approximately " << calc_pi
        << ". Error is " << fabs(calc_pi - actual_pi)
        << endl
        << "_____ "
        << endl;
    }
} //end else
if (rank == 0) { /*I am root node*/
    cout << "\nCompute with new intervals? (y/n)" << endl; cin >> resp1;
}
} //end while
MPI::Finalize(); //terminate MPI
return 0;
} //end main

```

```
//functions
void printit()
{
    cout << "\n*****" << endl
        << "Welcome to the pi calculator!" << endl
        << "Programmer: K. Spry" << endl
        << "You set the number of divisions \nfor estimating the integral:
\n\tf(x)=4/(1+x^2) "
        << endl
        << "*****" << endl;
} //end printit
```

## Gravitational *N*-Body Problem

Finding positions and movements of bodies in space subject to gravitational forces from other bodies, using Newtonian laws of physics.

## Gravitational $N$ -Body Problem Equations

Gravitational force between two bodies of masses  $m_a$  and  $m_b$  is:

$$F = \frac{Gm_a m_b}{r^2}$$

$G$  is the gravitational constant and  $r$  the distance between the bodies. Subject to forces, body accelerates according to Newton's 2nd law:

$$F = ma$$

$m$  is mass of the body,  $F$  is force it experiences, and  $a$  the resultant acceleration.

## Details

Let the time interval be  $t$ . For a body of mass  $m$ , the force is:

$$F = \frac{m(v^{t+1} - v^t)}{t}$$

New velocity is:

$$v^{t+1} = v^t + \frac{Ft}{m}$$

where  $v^{t+1}$  is the velocity at time  $t + 1$  and  $v^t$  is the velocity at time  $t$ .

Over time interval  $t$ , position changes by

$$x^{t+1} - x^t = v^t t$$

where  $x^t$  is its position at time  $t$ .

Once bodies move to new positions, forces change. Computation has to be repeated.

## Sequential Code

Overall gravitational  $N$ -body computation can be described by:

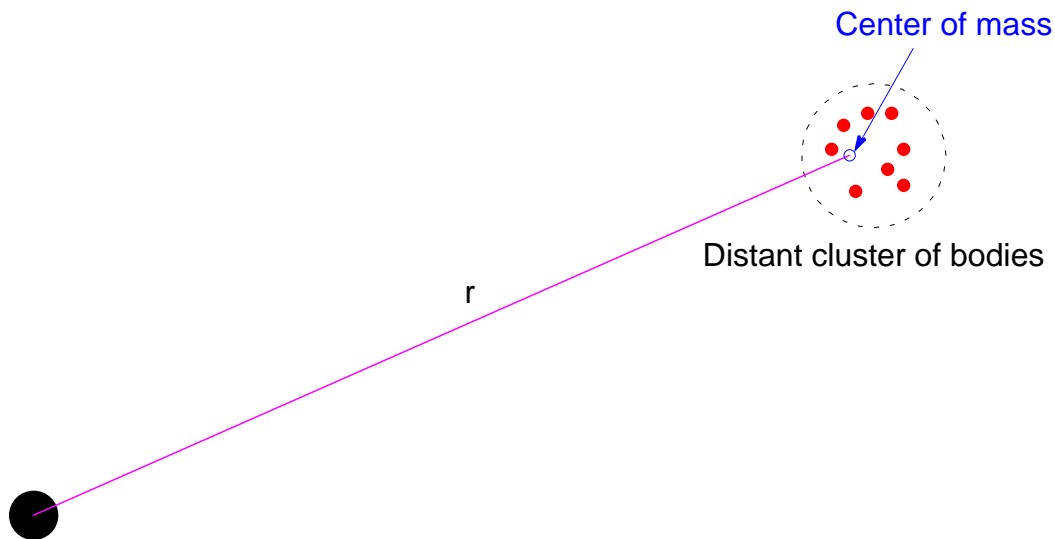
```
for (t = 0; t < tmax; t++)      /* for each time period */
  for (i = 0; i < N; i++) {    /* for each body */
    F = Force_routine(i);      /* compute force on ith body */
    v[i]_new = v[i] + F * dt / m; /* compute new velocity */
    x[i]_new = x[i] + v[i]_new * dt; /* and new position */
  }
for (i = 0; i < nmax; i++) {   /* for each body */
  x[i] = x[i]_new;             /* update velocity & position*/
  v[i] = v[i]_new;
}
```

## Parallel Code

The sequential algorithm is an  $O(N^2)$  algorithm (for one iteration) as each of the  $N$  bodies is influenced by each of the other  $N - 1$  bodies.

Not feasible to use this direct algorithm for most interesting  $N$ -body problems where  $N$  is very large.

Time complexity can be reduced using observation that a cluster of distant bodies can be approximated as a single distant body of the total mass of the cluster sited at the center of mass of the cluster:



## Barnes-Hut Algorithm

Start with whole space in which one cube contains the bodies (or particles).

- First, this cube is divided into eight subcubes.
- If a subcube contains no particles, the subcube is deleted from further consideration.
- If a subcube contains one body, this subcube retained
- If a subcube contains more than one body, it is recursively divided until every subcube contains one body.

Creates an *octtree* - a tree with up to eight edges from each node.

The leaves represent cells each containing one body.

After the tree has been constructed, the total mass and center of mass of the subcube is stored at each node.

Force on each body obtained by traversing tree starting at root, stopping at a node when the clustering approximation can be used, e.g. when:

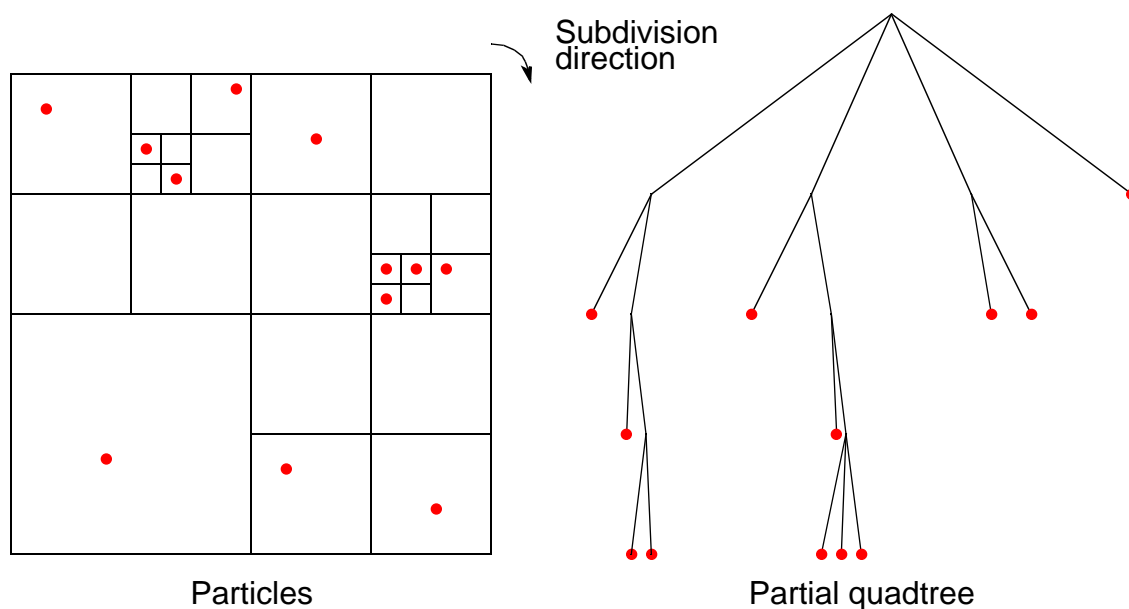
$$r \leq \alpha d$$

where  $\alpha$  is a constant typically 1.0 or less.

Constructing tree requires a time of  $O(n \log n)$ , and so does computing all the forces, so that the overall time complexity of the method is  $O(n \log n)$ .

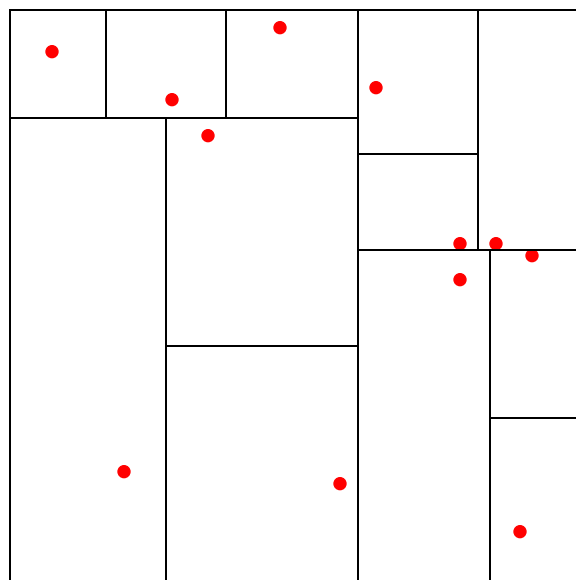


## Recursive division of two-dimensional space

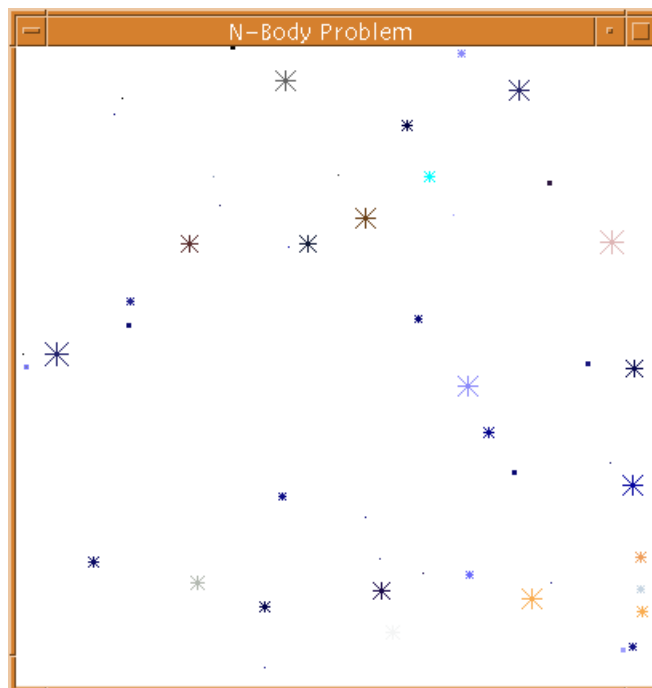


## Orthogonal Recursive Bisection

(For 2-dimensional area) First, a vertical line found that divides area into two areas each with equal number of bodies. For each area, a horizontal line found that divides it into two areas each with equal number of bodies. Repeated as required.



**Astrophysical  $N$ -body simulation by Scott Linssen (undergraduate UNCC student, 1997) using  $O(N^2)$  algorithm.**



**Astrophysical  $N$ -body simulation by David Messenger (UNCC student 1998) using Barnes-Hut algorithm.**

