Chapter 4

# 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.

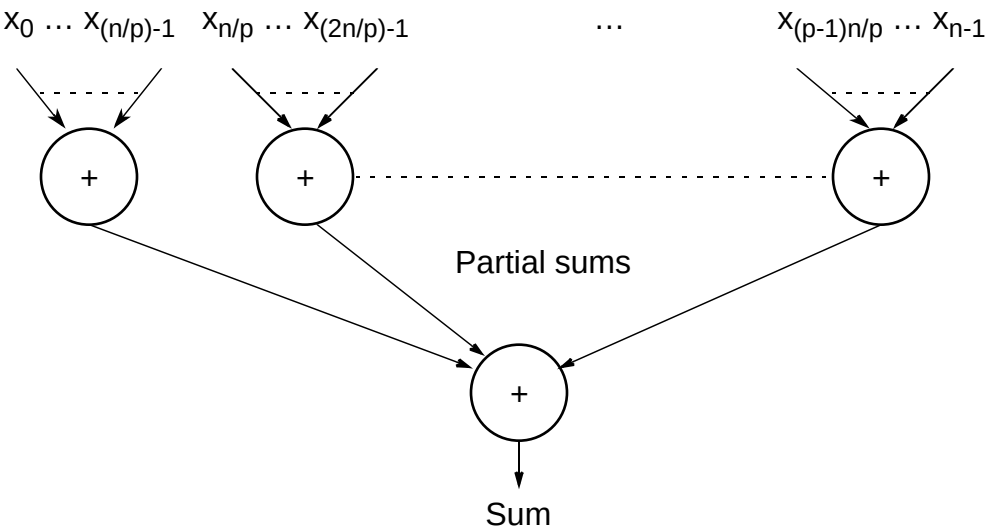
Recursive divide and conquer amenable to parallelization because separate processes can be used for divided parts.
Also usually data is naturally localized.

# 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

# Partitioning a sequence of numbers into parts and adding the parts



## Master

```
s = n/p;                                    /* number of numbers for slaves*/
for (i = 0, x = 0; i < p; i++, x = x + s)
   send(&numbers[x], s, P_i);               /* send s numbers to slave */

sum = 0;
for (i = 0; i < p; i++) {                   /* wait for results from slaves */
   recv(&part_sum, P_ANY);
   sum = sum + part_sum;                    /* accumulate partial sums */
}
```

## Slave

```
recv(numbers, s, P_master);                 /* receive s numbers from master */
part_sum = 0;
for (i = 0; i < s; i++)                     /* add numbers */
   part_sum = part_sum + numbers[i];
send(&part_sum, P_master);                  /* send sum to master */
```

## Master

```
s = n/p;                                    /* number of numbers for slaves */
bcast(numbers, s, P_slave_group);           /* send all numbers to slaves */
sum = 0;
for (i = 0; i < p; i++) {                   /* wait for results from slaves */
   recv(&part_sum, P_ANY);
   sum = sum + part_sum;                    /* accumulate partial sums */
}
```

## Slave

```
bcast(numbers, s, P_master);                /* receive all numbers from master*/
start = slave_number * s;                   /* slave number obtained earlier */
end = start + s;
part_sum = 0;
for (i = start; i < end; i++)               /* add numbers */
  part_sum = part_sum + numbers[i];
send(&part_sum, P_master);                  /* send sum to master */
```

## Master

```
s = n/p;                                          /* number of numbers */
scatter(numbers,&s,P_group,root=master);          /* send numbers to slaves */
reduce_add(&sum,&s,P_group,root=master);          /* results from slaves */
```

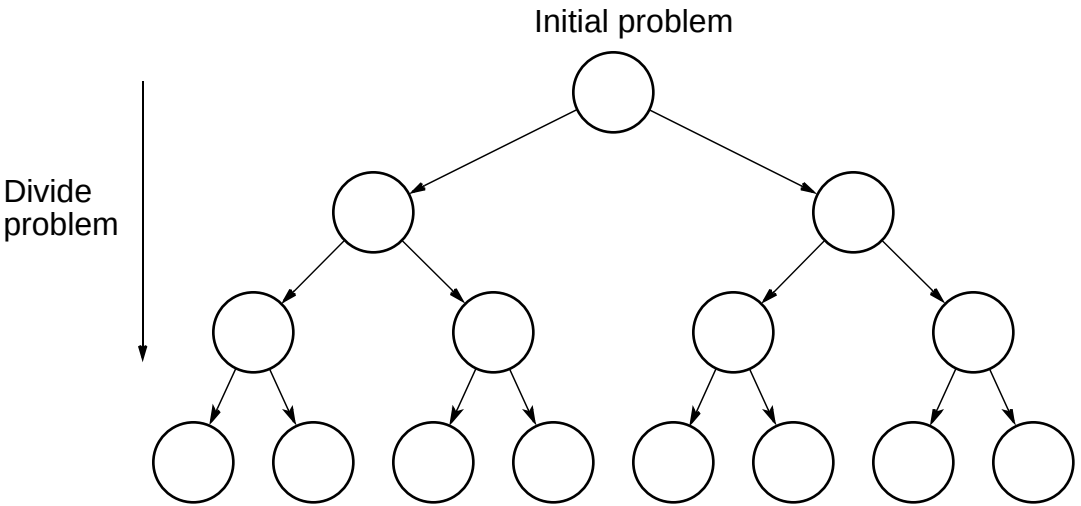
## Slave

```
scatter(numbers,&s,P_group,root=master);          /* receive s numbers */
.                                                 /* add numbers */
.
.
reduce_add(&part_sum,&s,P_group,root=master);     /* send sum to master */
```

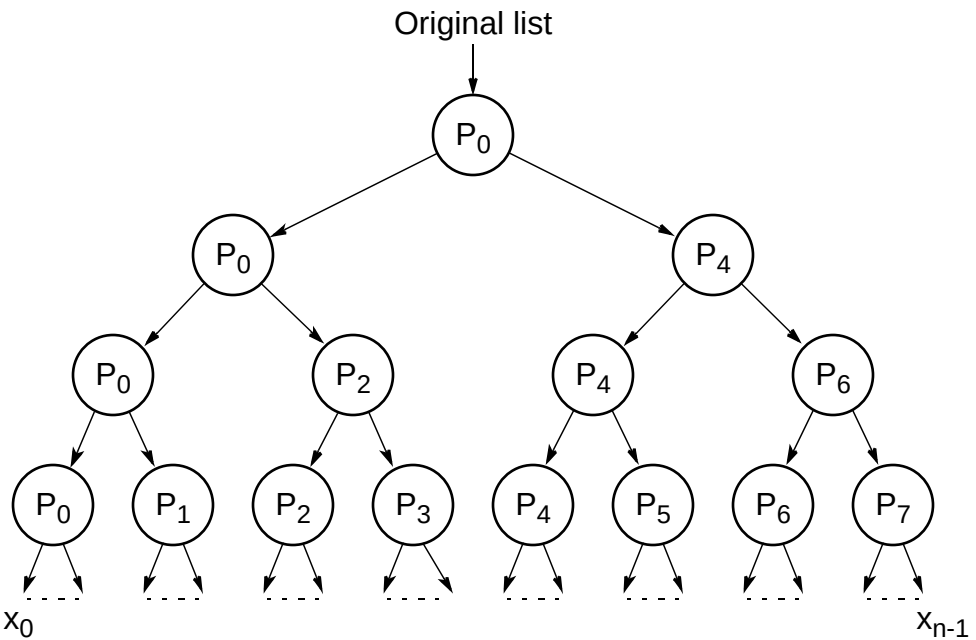
A sequential recursive definition for adding a list of numbers is[1]

```
int add(int *s)                                    /* add list of numbers, s */
{
   if (number(s) <= 2) return (n1 + n2);           /* see explanation */
   else {
      Divide (s, s1, s2);          /* divide s into two parts, s1 and s2 */
      part_sum1 = add(s1);         /*recursive calls to add sub lists */
      part_sum2 = add(s2);
      return (part_sum1 + part_sum2);
   }
}
```

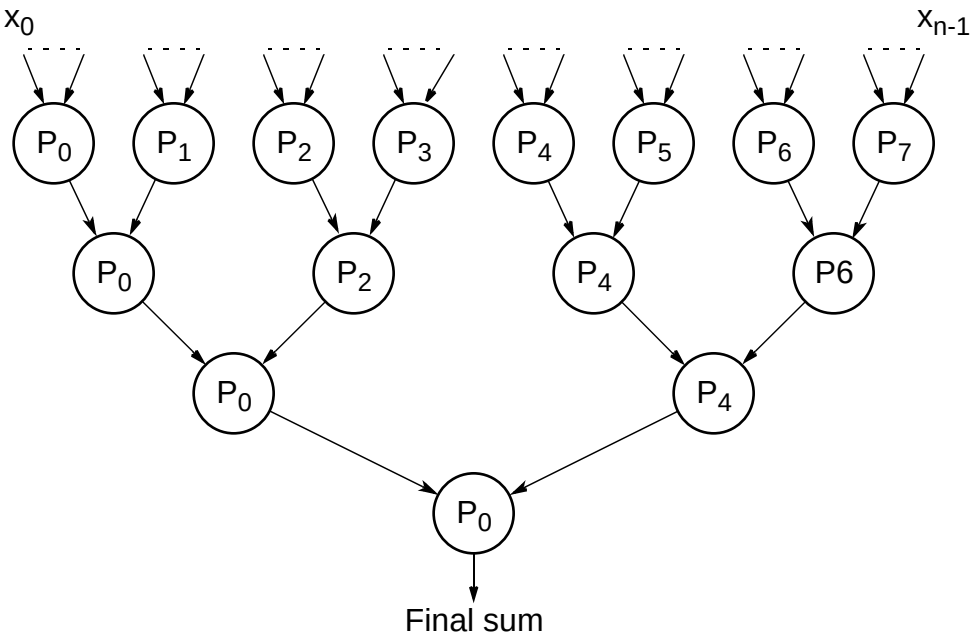
# Tree construction



# Dividing a list into parts



# Partial summation



Process $P_0$

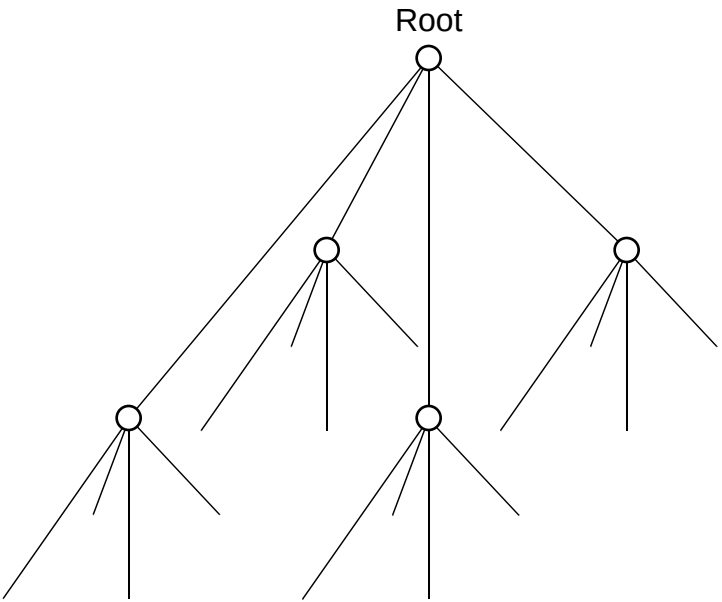
```
                                        /* division phase */
divide(s1, s1, s2);                     /* divide s1 into two, s1 and s2 */
send(s2, P4);                           /* send one part to another process */
divide(s1, s1, s2);
send(s2, P2);
divide(s1, s1, s2);
send(s2, P1};
part_sum = *s1;                         /* combining phase */
recv(&part_sum1, P1);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P2);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P4);
part_sum = part_sum + part_sum1;
```

The code for process $P_4$ might take the form

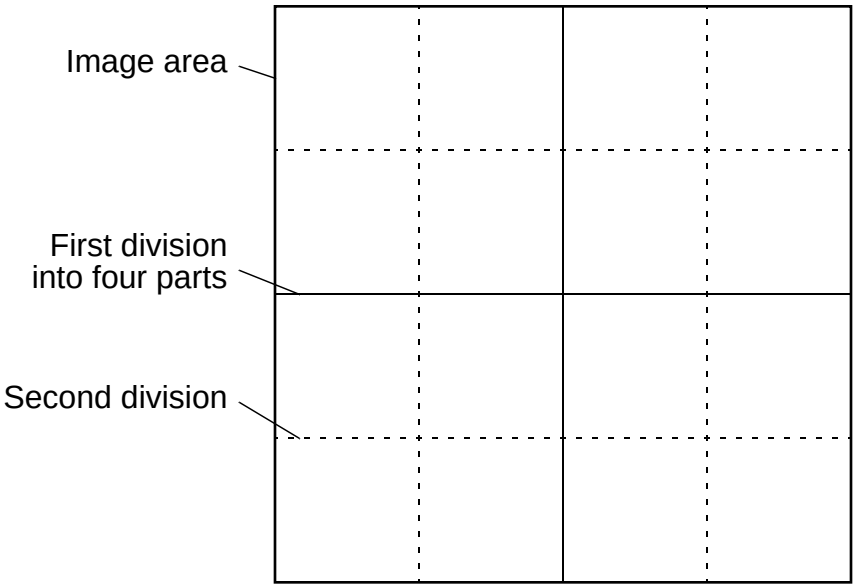
Process $P_4$

```
recv(s1, P0);                          /* division phase */
divide(s1, s1, s2);
send(s2, P6);
divide(s1, s1, s2);
send(s2, P5);
part_sum = *s1;                        /* combining phase */
recv(&part_sum1, P5);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P6);
part_sum = part_sum + part_sum1;
send(&part_sum, P0);
```

# Quadtree



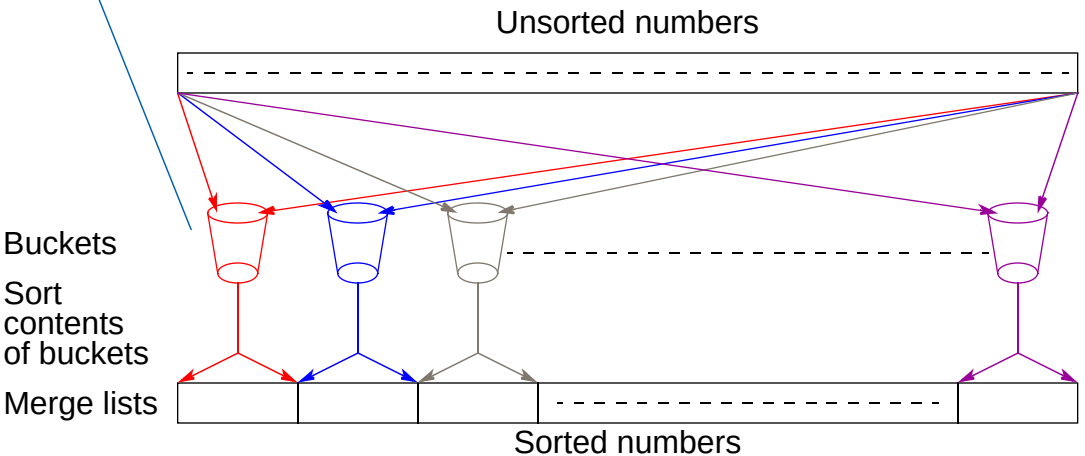
# Dividing an image



```
int add(int *s)                          /* add list of numbers, s */
{
   if (number(s) =< 4) return(n1 + n2 + n3 + n4);
   else {
      Divide (s,s1,s2,s3,s4);            /* divide s into s1,s2,s3,s4*/
      part_sum1 = add(s1);               /*recursive calls to add sublists */
      part_sum2 = add(s2);
      part_sum3 = add(s3);
      part_sum4 = add(s4);
      return (part_sum1 + part_sum2 + part_sum3 + part_sum4);
   }
}
```

# Bucket sort

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

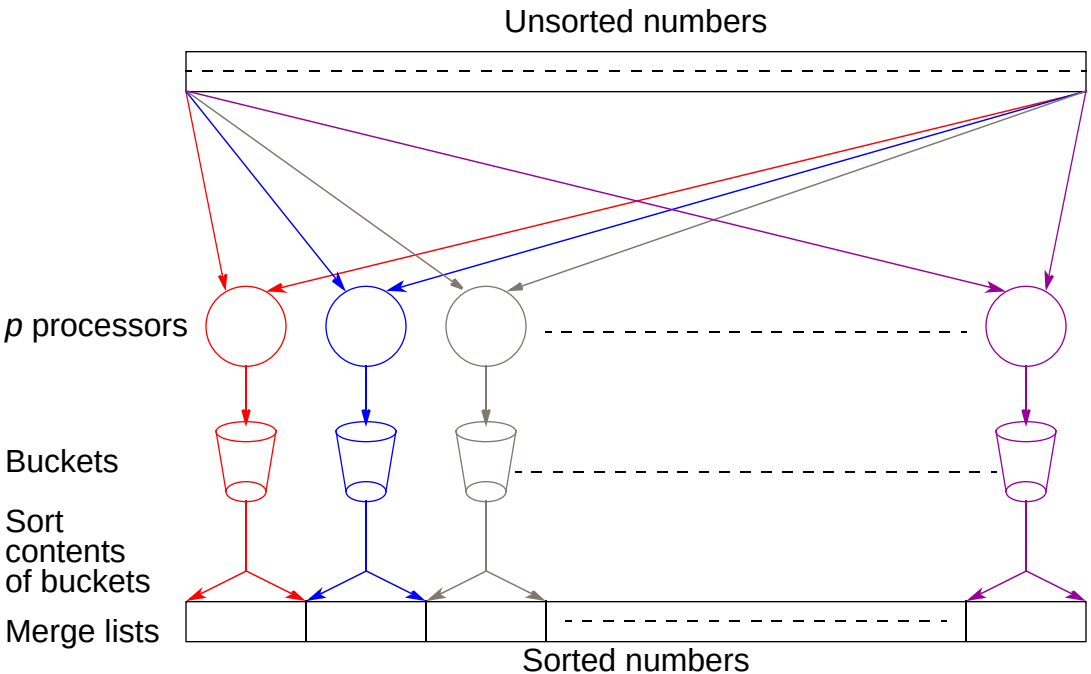



Sequental sorting time complexity: O(n log(n/m).
Works well if the original numbers uniformly distributed across a known interval, say 0 to *a* − 1.

# Parallel version of bucket sort

## Simple approach

Assign one processor for each bucket.



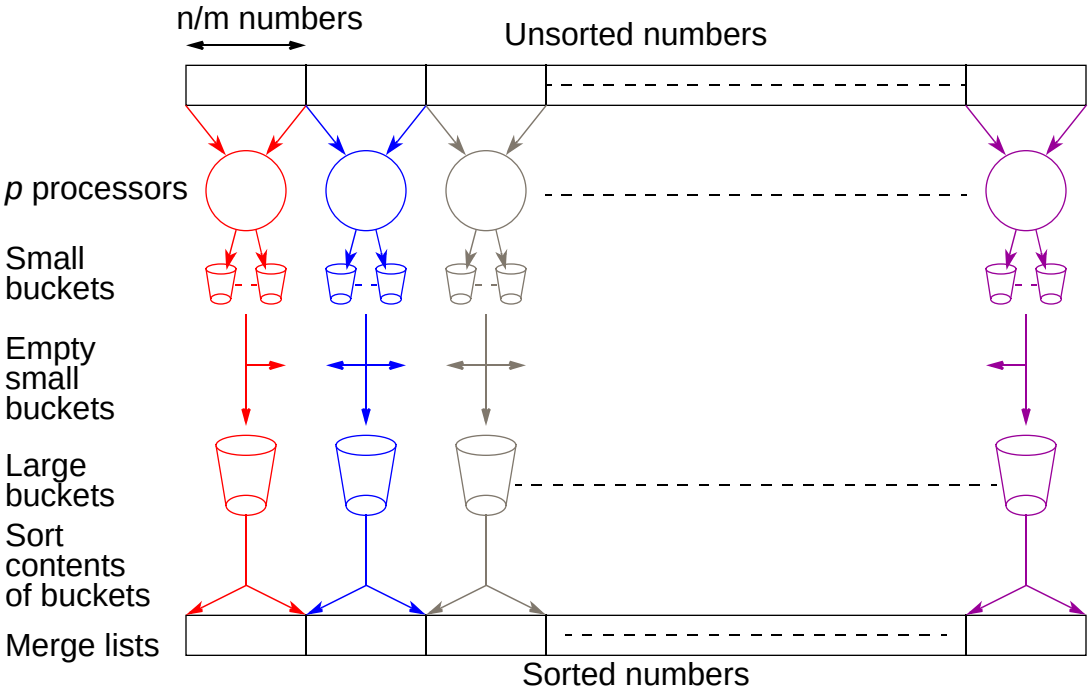
# Further Parallelization

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

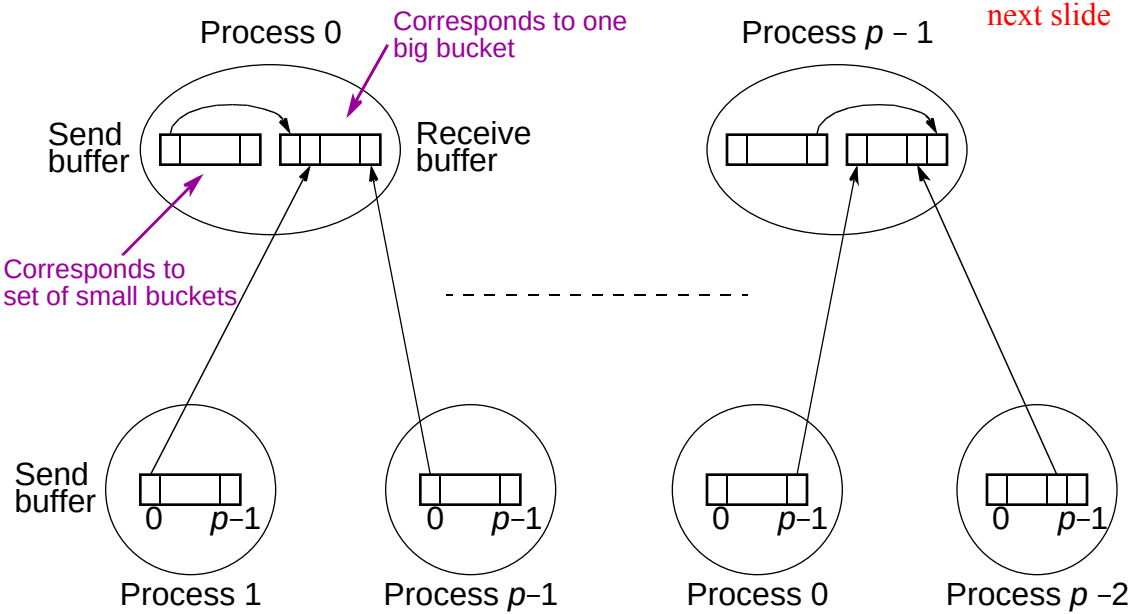



Introduces new message-passing operation - all-to-all broadcast.

# "all-to-all" broadcast routine

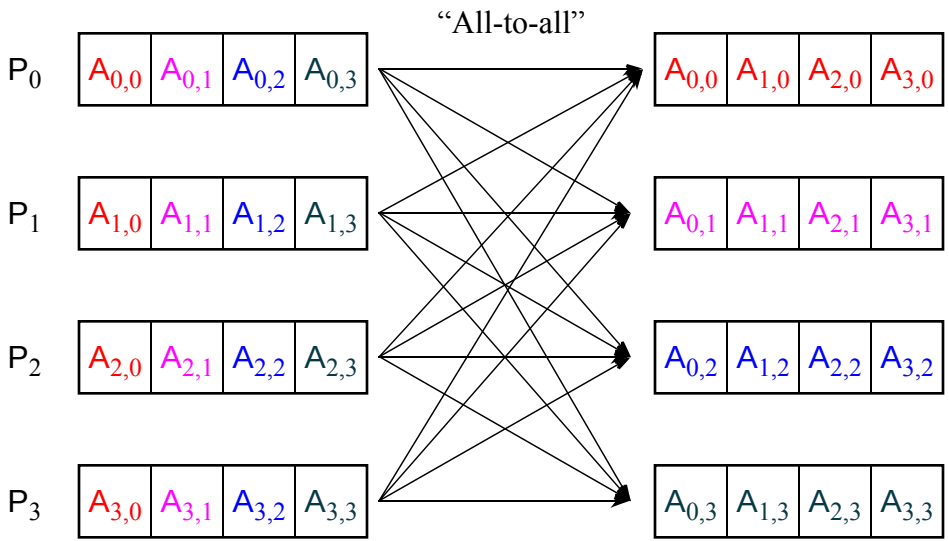
Sends data from each process to every other process

See also next slide



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

Tranposes a matrix.

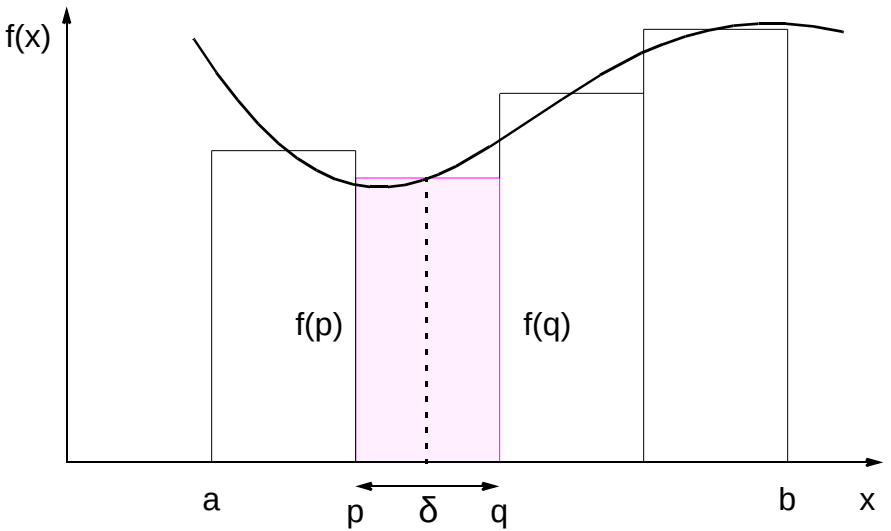



**Effect of "all-to-all" on an array**

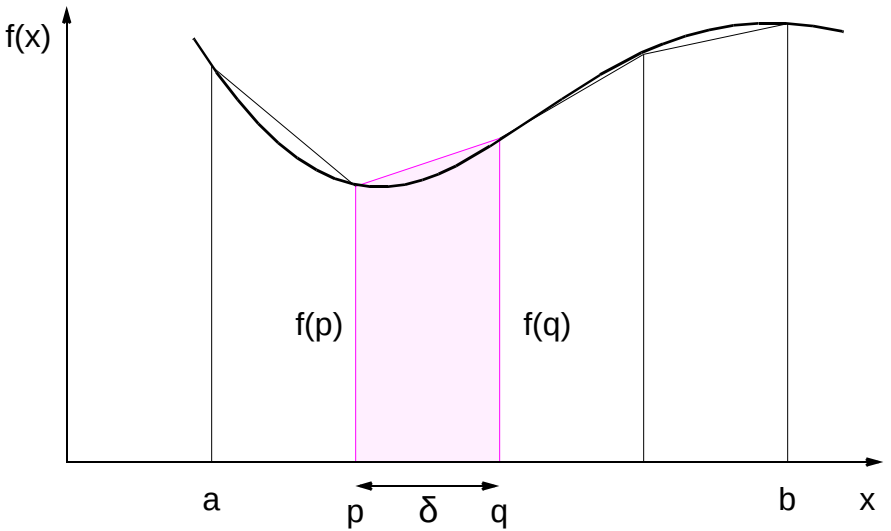
# Numerical integration using rectangles.

Each region calculated using an approximation given by rectangles:

Aligning the rectangles:



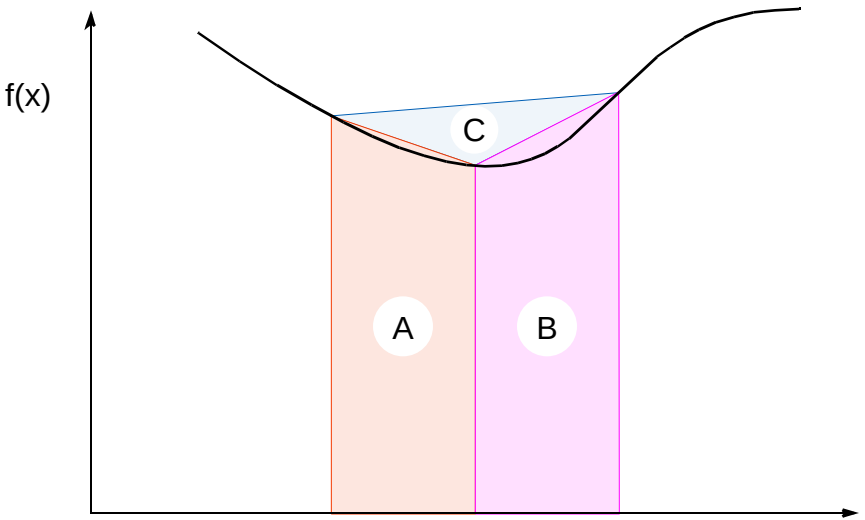
# Numerical integration using trapezoidal method




May not be better!

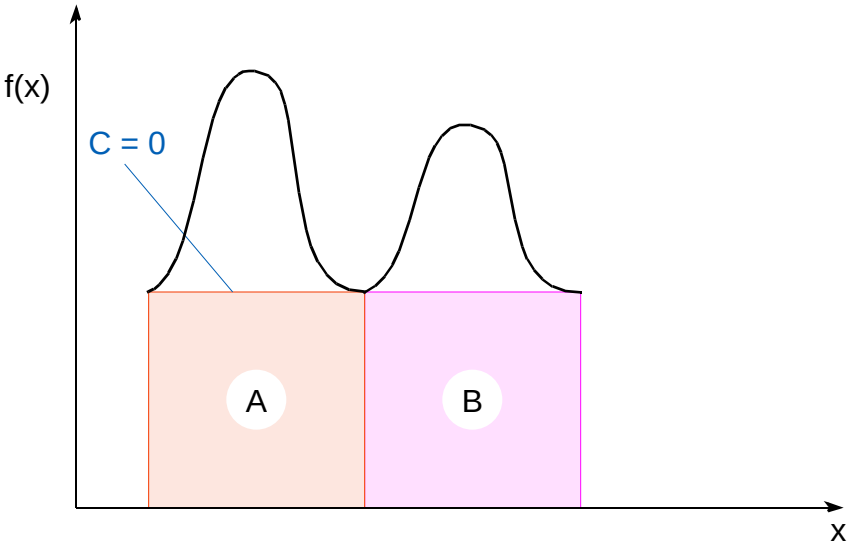
# Adaptive Quadrature

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 π

## 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 = 3.14159265358979323846264 3;   //for comparison later
int n, rank, num_proc, i;
double temp_pi, calc_pi, int_size, part_sum, x;
char response = 'y', resp1 = '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 (resp1 == '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 $\Delta t$. For a body of mass $m$, the force is:

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

New velocity is:

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

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

Over time interval $\Delta t$, position changes by

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

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:

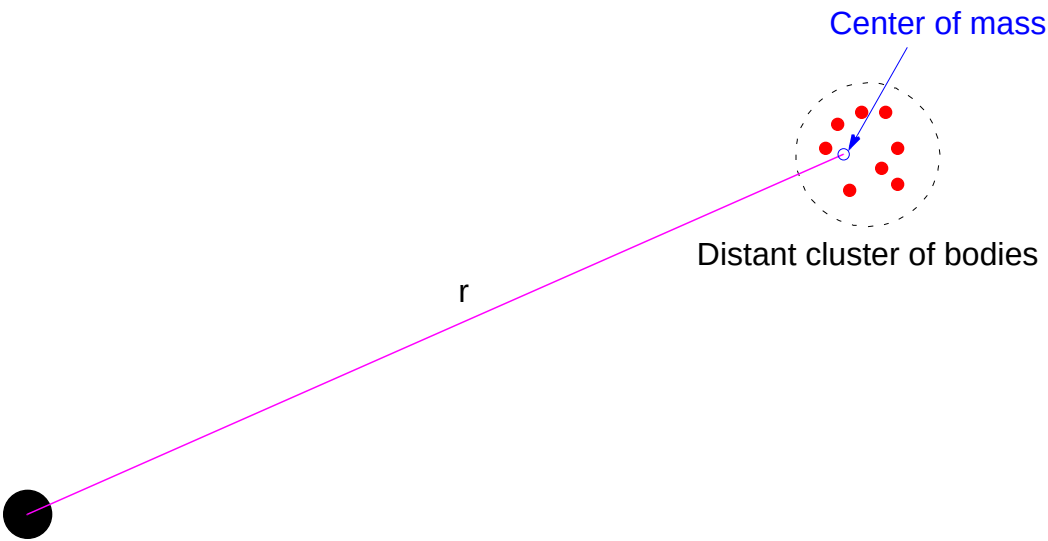
```
for (t = 0; t < tmax; t++)     /* for each time period */
  for (i = 0; i < N; i++) {    /* for each body */
    F = Force_routine(i);      /* compute force on ith body */
    v[i]new = v[i] + F * dt / m;  /* compute new velocity */
    x[i]new = x[i] + v[i]new * dt;  /* and new position */
  }
for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

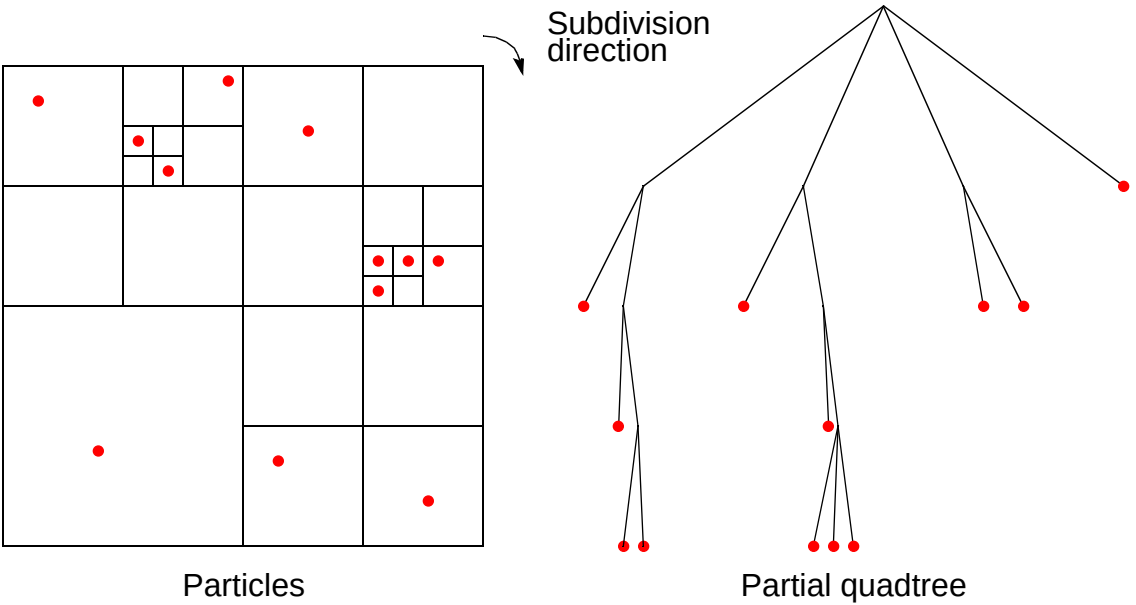
Force on each body obtained by traversing tree starting at root, stopping at a node when the clustering approximation can be used, e.g. when:

$$r \geq \frac{d}{\theta}$$

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

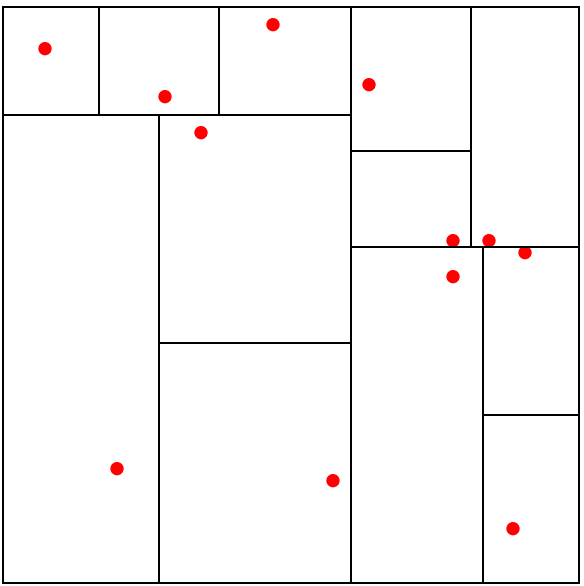
Constructing tree requires a time of O($n$log$n$), and so does computing all the forces, so that the overall time complexity of the method is O($n$log$n$).

# Recursive division of two-dimensional space

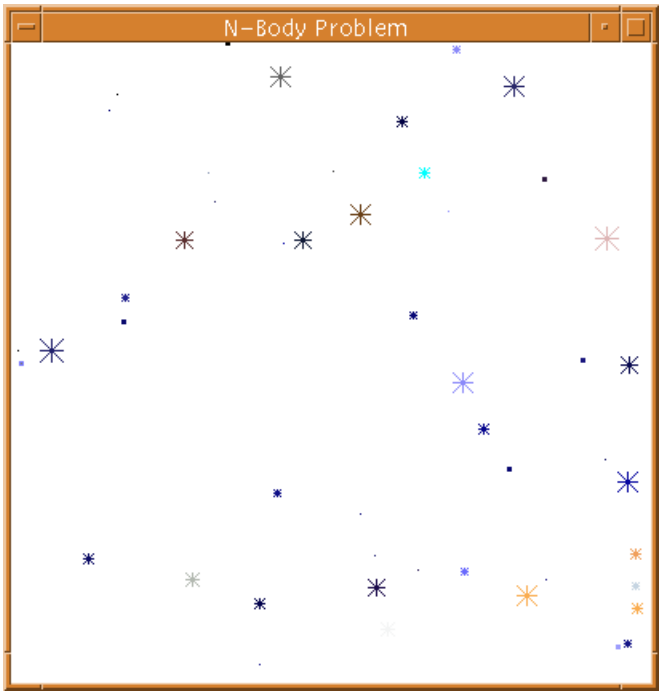


# Orthogonal Recursive Bisection

(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 Messager (UNCC student 1998) using Barnes-Hut algorithm.