Partitioning and Divide-and-Conquer Strategies

Partitioning Strategies

Partitioning simply divides the problem into parts

Example - Adding a sequence of numbers

We might consider dividing the sequence into m parts of n/m numbers each, ($x_0 \ldots x_{(n/m)-1}$), ($x_{n/m} \ldots x_{(2n/m)-1}$), …, ($x_{(m-1)n/m} \ldots x_{n-1}$), at which point m processors (or processes) can each add one sequence independently to create partial sums.

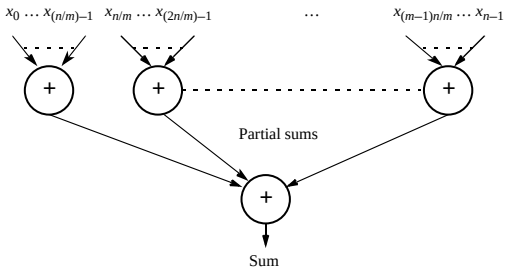

Figure 4.1 Partitioning a sequence of numbers into parts and adding the parts.

Using separate `send()`s and `recv()`s

Master

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

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

Slave

```
recv(numbers, s, Pmaster);        /* 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, Pmaster);         /* send sum to master */
```

Using Broadcast/multicast Routine

Master

```
s = n/m;                          /* number of numbers for slaves */
bcast(numbers, s, P_slave_group);     /* send all numbers to slaves */
sum = 0;
for (i = 0; i < m; 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 */
```


Using scatter and reduce routines

Master

```
s = n/m;                                  /* 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 */
```

Analysis

Sequential

Requires $n - 1$ additions with a time complexity of $O(n)$.

Parallel

Using individual send and receive routines

Phase 1 — Communication

$$t_{comm1} = m(t_{startup} + (n/m)t_{data})$$

Phase 2 — Computation

$$t_{comp1} = n/m - 1$$

Phase 3 — Communication

Returning partial results using individual send and receive routines

$$t_{comm2} = m(t_{startup} + t_{data})$$

Phase 4 — Computation

Final accumulation

$$t_{comp2} = m - 1$$

Overall

$$t_p = (t_{comm1} + t_{comm2}) + (t_{comp1} + t_{comp2})$$

$$= 2mt_{startup} + (n + m)t_{data} + m + n/m - 2$$

or

$$t_p = O(n + m)$$

We see that the parallel time complexity is worse than the sequential time complexity.

Divide and Conquer

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

A sequential recursive definition for adding a list of numbers is

```
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);
  }
}
```

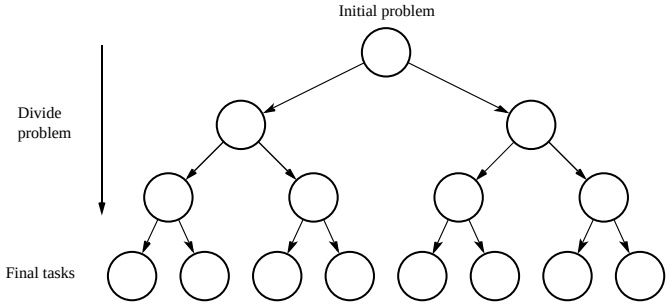

Figure 4.2 Tree construction.

Parallel Implementation

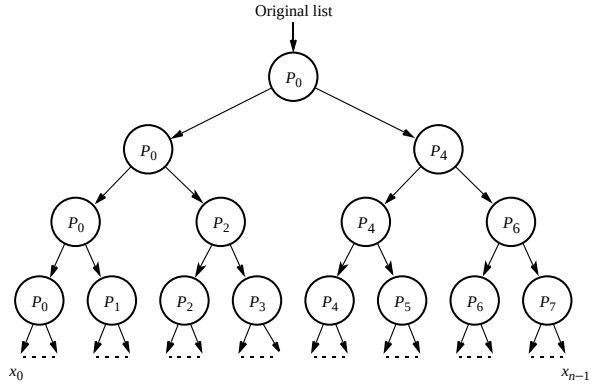

Figure 4.3 Dividing a list into parts.

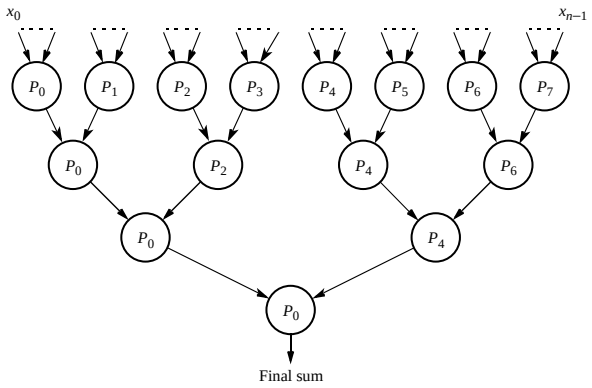

Figure 4.4 Partial summation.

Parallel Code

Suppose we statically create eight processors (or processes) to add a list of numbers.

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);
```

Similar sequences are required for the other processes.

Analysis

Assume that n is a power of 2. The communication setup time, $t_{startup}$, is not included in the following for simplicity.

Communication

Division phase

$$t_{comm1} = \frac{n}{2}t_{data} + \frac{n}{4}t_{data} + \frac{n}{8}t_{data} + \ldots + \frac{n}{p}t_{data} = \frac{n(p-1)}{p}t_{data}$$

Combining phase

$$t_{comm2} = t_{data}\log p$$

Total communication time

$$t_{comm} = t_{comm1} + t_{comm2} = \frac{n(p-1)}{p}t_{data} + t_{data}\log p$$

Computation

$$t_{comp} = \frac{n}{p} + \log p$$

Total Parallel Execution Time

$$t_p = \frac{n(p-1)}{p}t_{data} + t_{data}\log p + \frac{n}{p} + \log p$$

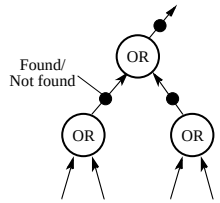

Figure 4.5 Part of a search tree.

M-ary Divide and Conquer

Divide and conquer can also be applied where a task is divided into more than two parts at each stage.

For example, if the task is broken into four parts, the sequential recursive definition would be

```
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);
  }
}
```

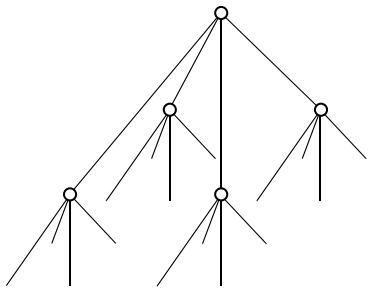

Figure 4.6 Quadtree.

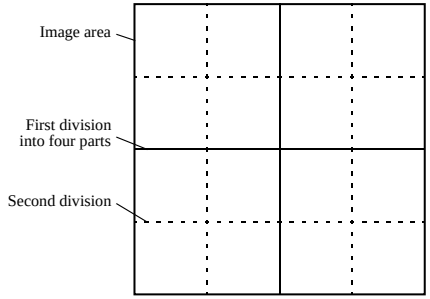

Figure 4.7 Dividing an image.

Divide-and-Conquer Examples

Sorting Using Bucket Sort

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

This interval is divided into m equal regions, 0 to $a/m - 1$, a/m to $2a/m - 1$, $2a/m$ to $3a/m - 1$, … and one "bucket" is assigned to hold numbers that fall within each region.

The numbers are simply placed into the appropriate buckets.

The numbers in each bucket will be sorted using a sequential sorting algorithm

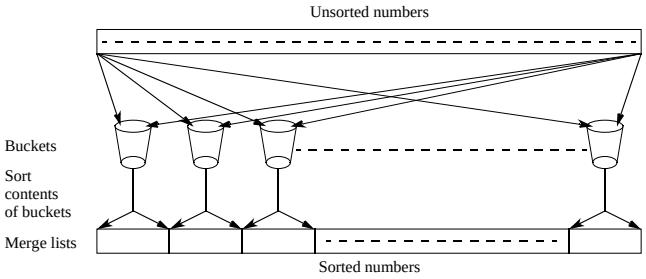

Figure 4.8 Bucket sort.

Sequential time

$$t_s = n + m((n/m)\log(n/m)) = n + n\log(n/m) = O(n\log(n/m))$$

Parallel Algorithm

Bucket sort can be parallelized by assigning one processor for each bucket, which reduces the second term in the preceding equation to $(n/p)\log(n/p)$ for p processors (where $p = m$).

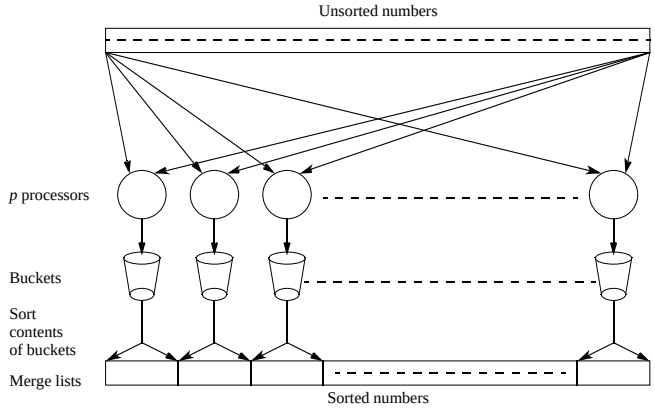

Figure 4.9 One parallel version of bucket sort.

Further Parallelization

By partitioning the 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.

These small buckets are then "emptied" into the 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).

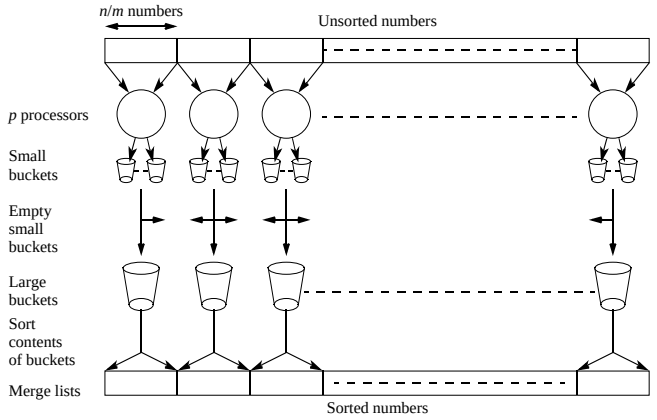

Figure 4.10 Parallel version of bucket sort.

Analysis

The following phases are needed:

1. Partition numbers.
2. Sort into small buckets.
3. Send to large buckets.
4. Sort large buckets.

Phase 1 — Computation and Communication

$$t_{comp1} = n$$

$$t_{comm1} = t_{startup} + t_{data}n$$

Phase 2 — Computation

$$t_{comp2} = n/p$$

Phase 3 — Communication.

If all the communications could overlap:

$$t_{comm3} = (p-1)(t_{startup} + (n/p^2)t_{data})$$

Phase 4 — Computation

$$t_{comp4} = (n/p)\log(n/p)$$

Overall

$$t_p = t_{startup} + t_{data}n + n/p + (p-1)(t_{startup} + (n/p^2)t_{data}) + (n/p)\log(n/p)$$

It is assumed that the numbers are uniformly distributed to obtain these formulas. The worst-case scenario would occur when all the numbers fell into one bucket!

“all-to-all” routine

For Phase 3 - sends data from each process to every other process

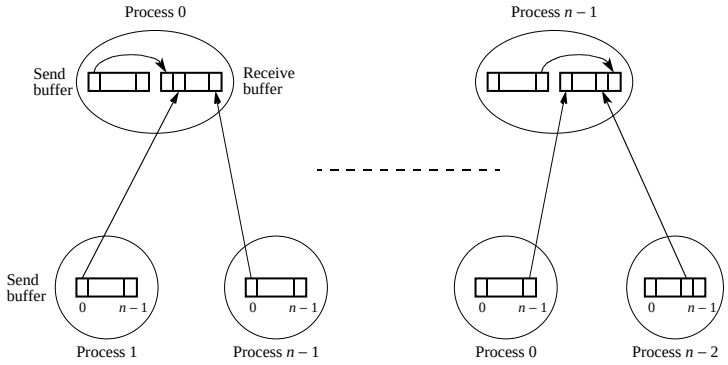

Figure 4.11 “All-to-all” broadcast.

The “all-to-all” routine will actually transfer the rows of an array to columns:

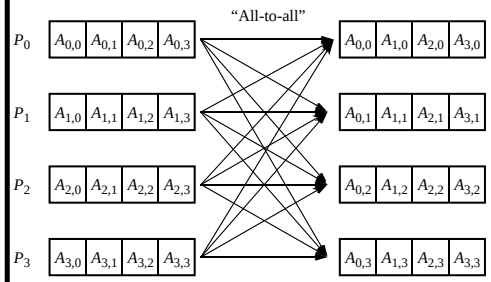

Figure 4.12 Effect of “all-to-all” on an array.

Numerical Integration

A general divide-and-conquer technique divides the region continually into parts and lets some optimization function decide when certain regions are sufficiently divided.

Example: numerical integration:

$$I = \int_a^b f(x)\,dx$$

Can divide the area into separate parts, each of which can be calculated by a separate process. Each region could be calculated using an approximation given by rectangles:

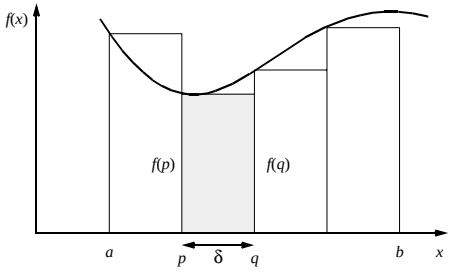

Figure 4.13 Numerical integration using rectangles.

A Better Approximation

Aligning the rectangles:

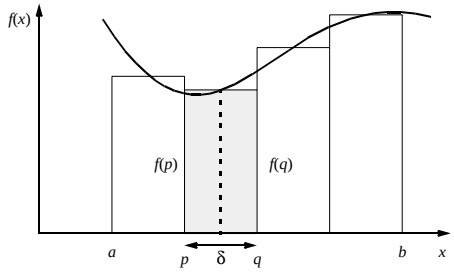

Figure 4.14 More accurate numerical integration using rectangles.

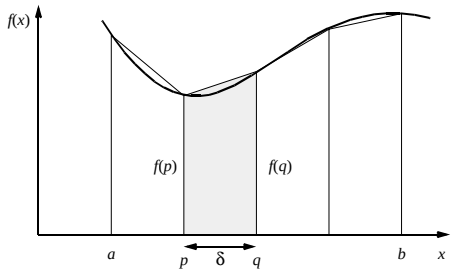

Figure 4.15 Numerical integration using the trapezoidal method.

Static Assignment

SPMD pseudocode:

Process P_i

```
if (i == master) {           /* read number of intervals required */
  printf("Enter number of intervals ");
  scanf(%d",&n);
}
bcast(&n, P_group);          /* broadcast interval to all processes */
region = (b - a)/p;          /* length of region for each process */
start = a + region * i;      /* starting x coordinate for process */
end = start + region;        /* ending x coordinate for process */
d = (b - a)/n;                           /* size of interval */
area = 0.0;
for (x = start; x < end; x = x + d)
  area = area + f(x) + f(x+d);
area = 0.5 * area * d;
reduce_add(&integral, &area, P_group);   /* form sum of areas */
```

A reduce operation is used to add the areas computed by the individual processes.

Can simplify the calculation somewhat by algebraic manipulation (see text).

Adaptive Quadrature

Method whereby the solution adapts to the shape of the curve

Example- use three areas, *A*, *B*, and *C*. The computation is terminated when the area computed for the largest of the *A* and *B* regions is sufficiently close to the sum of the areas computed for the other two regions.

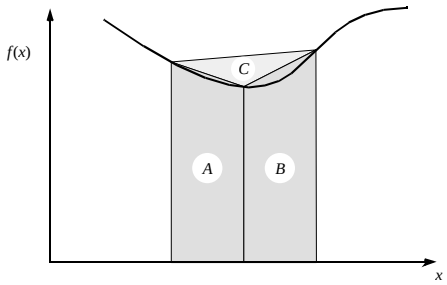

Figure 4.16 Adaptive quadrature construction.

Some care might be needed in choosing when to terminate.

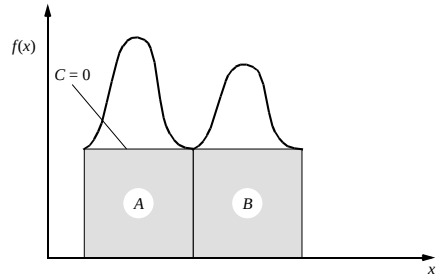

Figure 4.17 Adaptive quadrature with false termination.

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

Gravitational *N*-Body Problem

The objective is to find the positions and movements of the bodies in space (say planets) that are subject to gravitational forces from other bodies using Newtonian laws of physics.

The gravitational force between two bodies of masses m_a and m_b is given by

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

where G is the gravitational constant and r is the distance between the bodies.

Subject to forces, a body will accelerate according to Newton's second law:

$$F = ma$$

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

Let the time interval be Δt. Then, for a particular body of mass m, the force is given by

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

and a new velocity

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

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

If a body is moving at a velocity v over the time interval Δt, its 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, the forces change and the computation has to be repeated.

Three-Dimensional Space

Since the bodies are in a three-dimensional space, all values are vectors and have to be resolved into three directions, x, y, and z.

In a three-dimensional space having a coordinate system (x, y, z), the distance between the bodies at (x_a, y_a, z_a) and (x_b, y_b, z_b) is given by

$$r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2 + (z_b - z_a)^2}$$

The forces are resolved in the three directions, using, for example,

$$F_x = \frac{Gm_a m_b}{r^2}\left(\frac{x_b - x_a}{r}\right)$$

$$F_y = \frac{Gm_a m_b}{r^2}\left(\frac{y_b - y_a}{r}\right)$$

$$F_z = \frac{Gm_a m_b}{r^2}\left(\frac{z_b - z_a}{r}\right)$$

where the particles are of mass m_a and m_b and have the coordinates (x_a, y_a, z_a) and (x_b, y_b, z_b).

Sequential Code

The overall gravitational *N*-body computation can be described by the algorithm

```
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 and
    x[i]new = x[i] + v[i]new * dt;   /* new position (leap-frog) */
  }
for (i = 0; i < nmax; i++) {       /* for each body */
  x[i] = x[i]new;                  /* update velocity and position*/
  v[i] = v[i]new;
}
```


Parallel Code

The 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. It is not feasible to use this direct algorithm for most interesting *N*-body problems where *N* is very large.

The time complexity can be reduced using the 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:

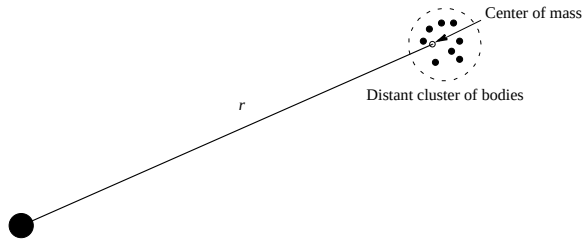

Figure 4.18 Clustering distant bodies.

Barnes-Hut Algorithm

Starts with the 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 more than one body, it is recursively divided until every subcube contains one body.

This process creates an *octtree*; that is, 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.

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

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

where θ is a constant typically 1.0 or less (θ is called the opening angle).

Constructing the 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 $O(n \log n)$.

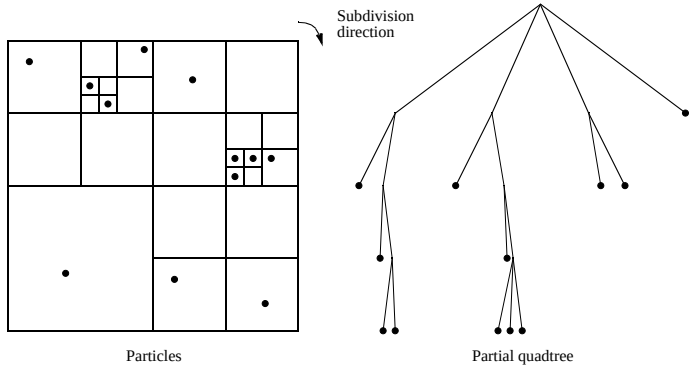

Figure 4.19 Recursive division of two-dimensional space.

Orthogonal Recursive Bisection

Example for a two-dimensional square area.

First, a vertical line is found that divides the area into two areas each with an equal number of bodies. For each area, a horizontal line is found that divides it into two areas each with an equal number of bodies. This is repeated until there are as many areas as processors, and then one processor is assigned to each area.

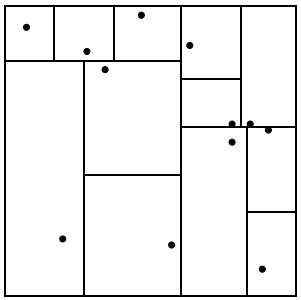

Figure 4.20 Orthogonal recursive bisection method.