Lesson 4: Partitioning and Divide-and-Conquer strategies

G Stefanescu — University of Bucharest

Parallel & Concurrent Programming
Fall, 2014

Partitioning strategies

Partitioning strategy is used as a generic term for any procedure based on the division of a problem into parts.

Example (adding numbers): The goal is to add a sequence of n numbers $x_0, \ldots, x_{n-1}$. The algorithm divides the sequence into p parts

$$\begin{array}{l} x_0, \ldots, x_{(n/p)-1} \\ x_{(n/p)}, \ldots, x_{2(n/p)-1} \\ \vdots \\ x_{(p-1)(n/p)}, \ldots, x_{p(n/p)-1} (= x_{n-1}) \end{array}$$

The numbers in each subsequence are added independently, then the resulting partial sums are added to get the final result.

..Partitioning strategies

1st implementation (using separate `send()` and `recv()`):

Master:

```
r = n/p
for (i=0,x=0; i<p; i++,x=x+r)
        send(numbers[x],r,$P_i$);
sum = 0;
for (i=0; i<p; i++){
        recv(partSum,$P_{any}$);
        sum = sum + partSum;
}
```

Slave:

```
recv(numbers,r,$P_{master}$);
partSum = 0;
for (i=0; i<r; i++)
        partSum = partSum + numbers[i];
send(partSum,$P_{master}$);
```

..Partitioning strategies

Analysis:

Sequential: Computation requires $n-1$ additions.
Hence time complexity is $O(n)$.

Parallel (1st implementation):

—1st communication: $t_{comm1} = p(t_{startup} + (n/p)t_{data})$
—computation (in slaves): $t_{comp1} = n/p - 1$
—2nd communication (return partial sum):

$$t_{comm1} = p(t_{startup} + t_{data})$$

—final computation: $t_{comp1} = p - 1$
—overall:

$$t_p = 2pt_{startup} + (n+p)t_{data} + n/p + p - 2$$

..Partitioning strategies

Conclusion:

- The computation part is *decreasing* from $n-1$ to $n/p+p-2$.
- The communication part is *linear* on both the size of data and the number of processes.
- The overall parallel time complexity is *worse* than sequential time complexity.
- To be useful, the slaves should have heavier computations (say, $t(n)$ such that the $t(n)-t(n/p) > 2pt_{startup}+(n+p)t_{data}$)

..Partitioning strategies

2nd implementation (using broadcast/multicast):

Master:

```
r = n/p
bcast(numbers,r,$P_{master}$,$P_{group}$);
sum = 0;
for (i=0; i<p; i++){
      recv(partSum,$P_{any}$);
      sum = sum + partSum;
}
```

Slave:

```
bcast(numbers,r,$P_{master}$,$P_{group}$);
start = slaveNumber * r;
end = start + r;
partSum = 0;
for (i=start; i<end; i++)
      partSum = partSum + numbers[i];
send(partSum,$P_{master}$);
```

..Partitioning strategies

3rd implementation (using scatter/reduce):

Master:

```
r = n/p
scatter(numbers,r,$P_{master}$,$P_{group}$);
reduceAdd(sum,$P_{master}$,$P_{group}$);
```

Slave:

```
scatter(numbers,r,$P_{master}$,$P_{group}$);
start = slaveNumber * r;
end = start + r;
partSum = 0;
for (i=start; i<end; i++)
      partSum = partSum + numbers[i];
reduceAdd(partSum,$P_{master}$,$P_{group}$);
```

Notice: The analysis of the 2nd or 3rd implementation is similar. (The real values depend on the network type and the particular implementations of broadcasting procedures.)

Divide-and-conquer

Divide-and-conquer is a particular partitioning strategy where *the subproblems are of the same form as the larger problem*.

Hence one can recursively apply the partition procedure to get smaller and smaller problems.

Add a list of numbers:

```
int add(list){
if (numbersOfElements(list) =< 2) return theirSum;
else{
      divide(list,list1,list2);
      return(add(list1) + add(list2));
     }
}
```

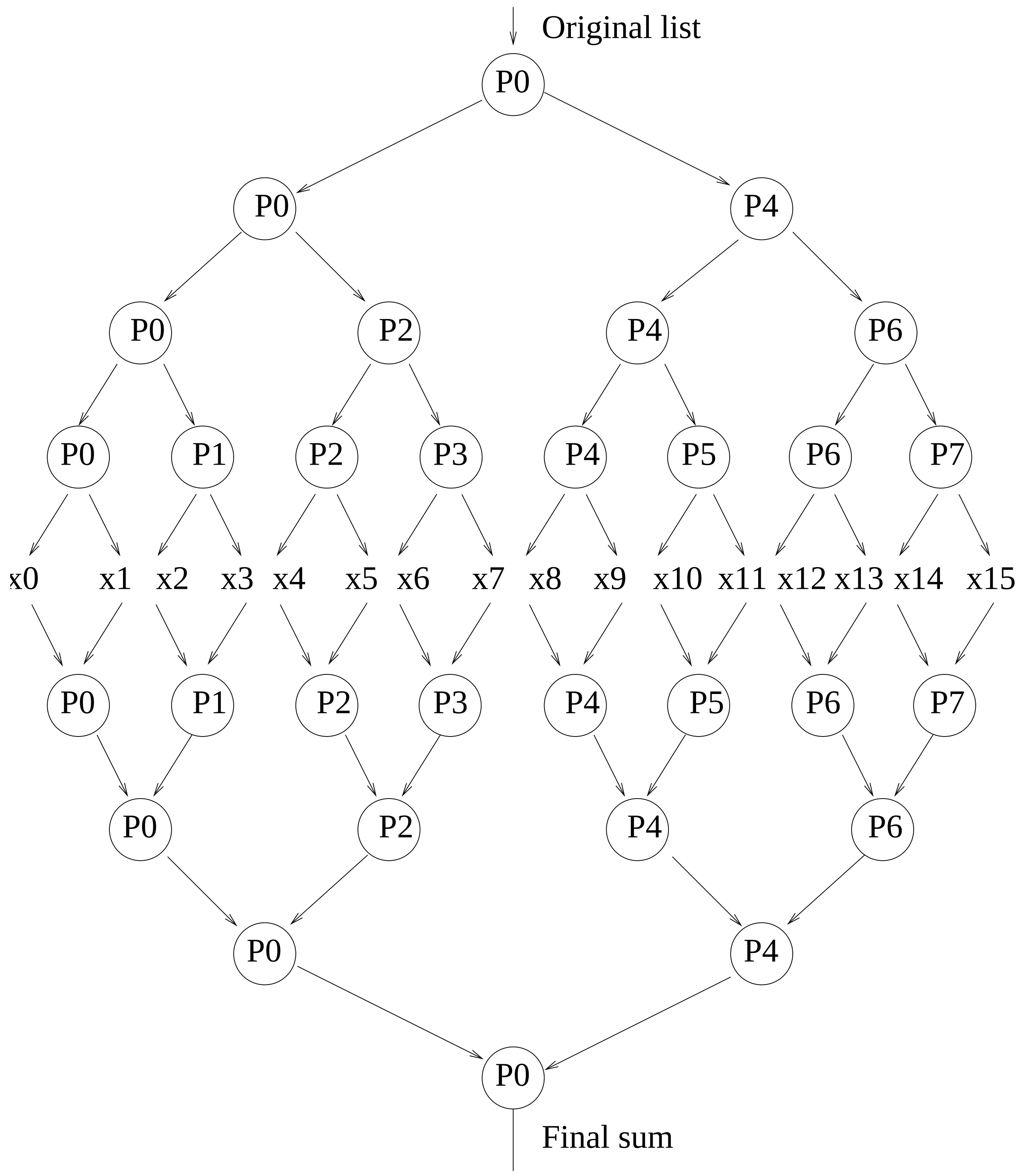
Notice: This is a general procedure which may be implemented either in a sequential or a parallel way.

..Divide-and-conquer

Parallel implementation:

(a) In a first stage, the list is recursively divided till each process receives a two-element list.

(b) In the next stage the partial results are collected using an opposite tree structure.



..Divide-and-conquer

Parallel code (Process P_i): Let

$$k(i) = \begin{cases} \bullet\ r \text{ — if } i = 0 \text{ (and there are } 2^r \text{ processes)} \\ \bullet \text{ the greatest power of 2 which divides } i \text{ — otherwise} \\ \quad [\text{i.e., } 2^{k(i)} \text{ divides } i, \text{ but } 2^{k(i)+1} \text{ does not divide } i] \end{cases}$$

```
if (!($myRank$ == 0)) recv(list,$P_{myRank-2^{k(i)}}$);
for(i=k-1; i>=0; i--){
    divide(list,list,list2);
    send(list2,$P_{myRank+2^i}$);
}
partSum = sumOf(list);
for(i=0; i<k; i++){
    recv(partSum2,$P_{myRank+2^i}$);
    partSum = partSum + partSum2;
}
if (!($myRank$ == 0)) send(partSum,$P_{myRank-2^{k(i)}}$);
```

..Divide-and-conquer

Examples:

Process P0

```
divide(list,list,list2);
send(list2,P4);
divide(list,list,list2);
send(list2,P2);
divide(list,list,list2);
send(list2,P1);
partSum = sumOf(List);
recv(partSum2,P1);
partSum = partSum + partSum2;
recv(partSum2,P2);
partSum = partSum + partSum2;
recv(partSum2,P4);
partSum = partSum + partSum2;
```

Process P4 ($k(4) = 2$):

```
recv(list,P0);
divide(list,list,list2);
send(list2,P6);
divide(list,list,list2);
send(list2,P5);
partSum = sumOf(List);
recv(partSum2,P5);
partSum = partSum + partSum2;
recv(partSum2,P6);
partSum = partSum + partSum2;
send(partSum,P0);
```

..Divide-and-conquer

Analysis: Assume $n = 2^k$ and $p = 2^r$. Then:

—division: $t_{comm1} = t_{startup}\log_2 p + (n/2 + n/4 + \ldots + n/(2^r))t_{data}$
$= t_{startup}\log_2 p + [n(p-1)/p]t_{data}$

—collecting results: $t_{comm2} = t_{startup}\log_2 p + t_{data}\log_2 p$

—computation: $t_{comp} = n/p - 1 + \log_2 p$

—overall:

$$t_p = 2t_{startup}\log_2 p + [\log_2 p + n(p-1)/p]t_{data} + n/p - 1 + \log_2 p$$

..Divide-and-conquer

Conclusion:

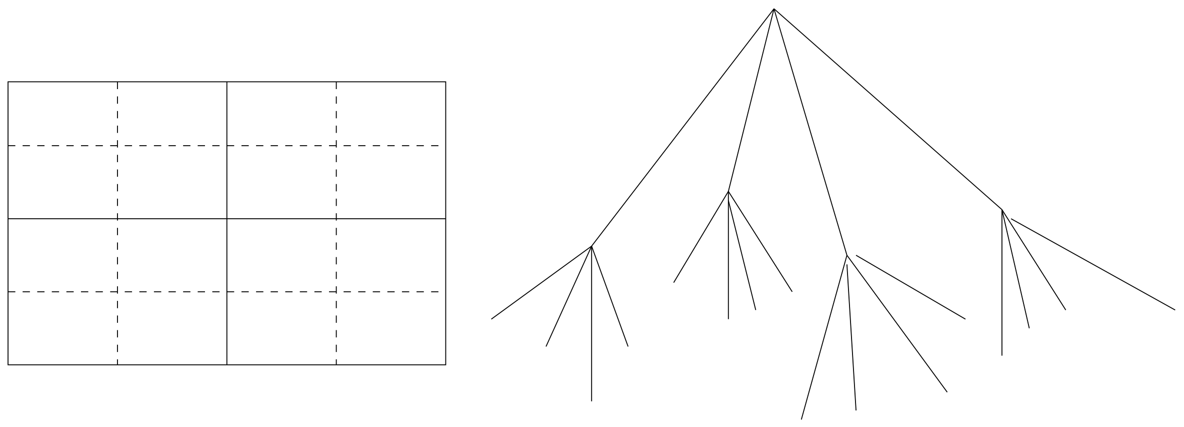
- The computation part is *decreasing*.
- The communication is still large: *linear* on the size of data, but now *logarithmic* on the number of processes.
- The algorithm becomes efficient when the computation part is large enough.

Notice that the work is not equally loaded on processes.

..Divide-and-conquer

M-ary divide and conquer: A similar approach may be used when the problem is divided into more than two parts at each stage [if possible]. Now, $m - ary$ trees are to be used, where m is the branching degree used at each stage.

Example: Such a technique may be applied to image processing by dividing each dimension into two parts at each stage, hence 4-ary trees are to be used [trees in which each node has four children.]



Case studies: 1. Sorting

Sorting using bucket sort:

1. The range of the numbers, say the interval $[min, max)$, is divided into p equal regions:

   $[min,\ min + (max - min)/p)$ (Range 1)

   $[min + (max - min)/p,\ min + 2(max - min)/p)$ (Range 2)

   $\vdots$

   $[min + (p-1)(max - min)/p,\ max)$ (Range p)

2. For each region a *bucket* is assigned to hold the numbers that fall within that region.
3. The numbers are placed into the appropriate buckets.
4. Finally, the numbers from each bucket are sorted using a sequential algorithm.

Notice: Works well if the numbers are uniformly distributed in $[min, max)$.

..Sorting

Sequential algorithm: Suppose there are n numbers uniformly distributed in $[min, max)$. Then each bucket has an average of n/p numbers. Hence

$$t_s = n + p(n/p)\log_2(n/p)$$

where:
- n — time to parse the numbers and place into buckets
- p — number of buckets
- $(n/p)\log_2(n/p)$ — sequential time to sort n/p numbers of each bucket

..Sorting

Parallel time (1st version): Sort the buckets in parallel. Hence

$$t1_p = n + (n/p)\log_2(n/p) + t_{startup} + nt_{data}$$

where, comparing with the former equation,

- p in the 2nd term disappears (work in parallel)
- $t_{startup} + nt_{data}$ is added (send data, via broadcast)

..Sorting

Parallel time (2nd version): A further parallelization is possible by *placing numbers into buckets in parallel*, as well. Suppose we have n numbers and p regions.

1. Separate the unsorted numbers $a_0, \ldots, a_{n-1}$ into groups of n/p elements: a_0 to $a_{n/p-1}$, $a_{n/p}$ to $a_{2n/p-1}$, ..., one for each processor $P_i (i = 1, \ldots, p)$.
2. Each processor P_i parses its group of n/p numbers and generates *small buckets* $b_{i1}, \ldots, b_{ip}$ (b_{ik} contains the numbers that fall into Range k).
3. The small buckets are sent to appropriate processors (P_i sends b_{ik} to P_k)
4. Each processor sorts its bucket of numbers.

..Sorting

Analysis: Assume we have n numbers, p processes and the numbers are uniformly distributed into buckets. Then:

1. send numbers to processes: $t_{comm1} = t_{startup} + nt_{data}$ (broadcast)
2. generate small buckets: $t_{comp1} = n/p$
3. send small buckets: $t_{comm2} = (p-1)(t_{startup} + (n/p^2)t_{data})$
(if communications could overlap; otherwise multiply by p)
4. sort large buckets: $t_{comp} = (n/p)\log_2(n/p)$

Overall:

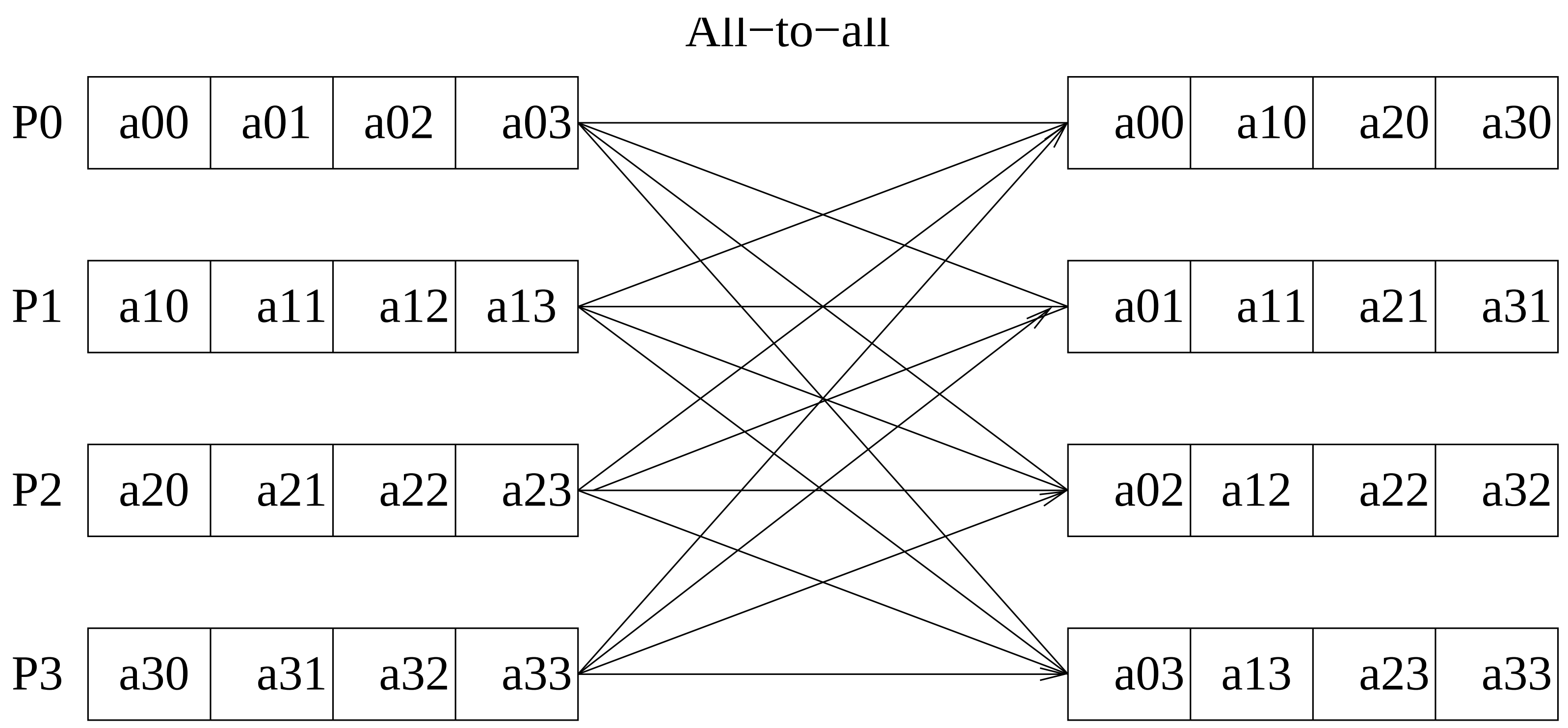
$$t_p = n/p + (n/p)\log_2(n/p) + pt_{startup} + (n + n/p^2)t_{data}$$

Conclusion: The computation time to generate buckets is decreasing from n to n/p, but the communication time is increasing.

..Sorting

All-to-all routines:

- A point where further improvement is possible is Step 3 where each process has to send specific data to all other processes.
- Such sort of communication may be made more efficient using standard routines, e.g., `MPI_Alltoall()`.



Case studies: 2. Numerical integration

Numerical integration:

- A general divide-and-conquer technique divides the region continuously into parts.
- The procedure stops when some optimization function decides that the regions are sufficiently divided.

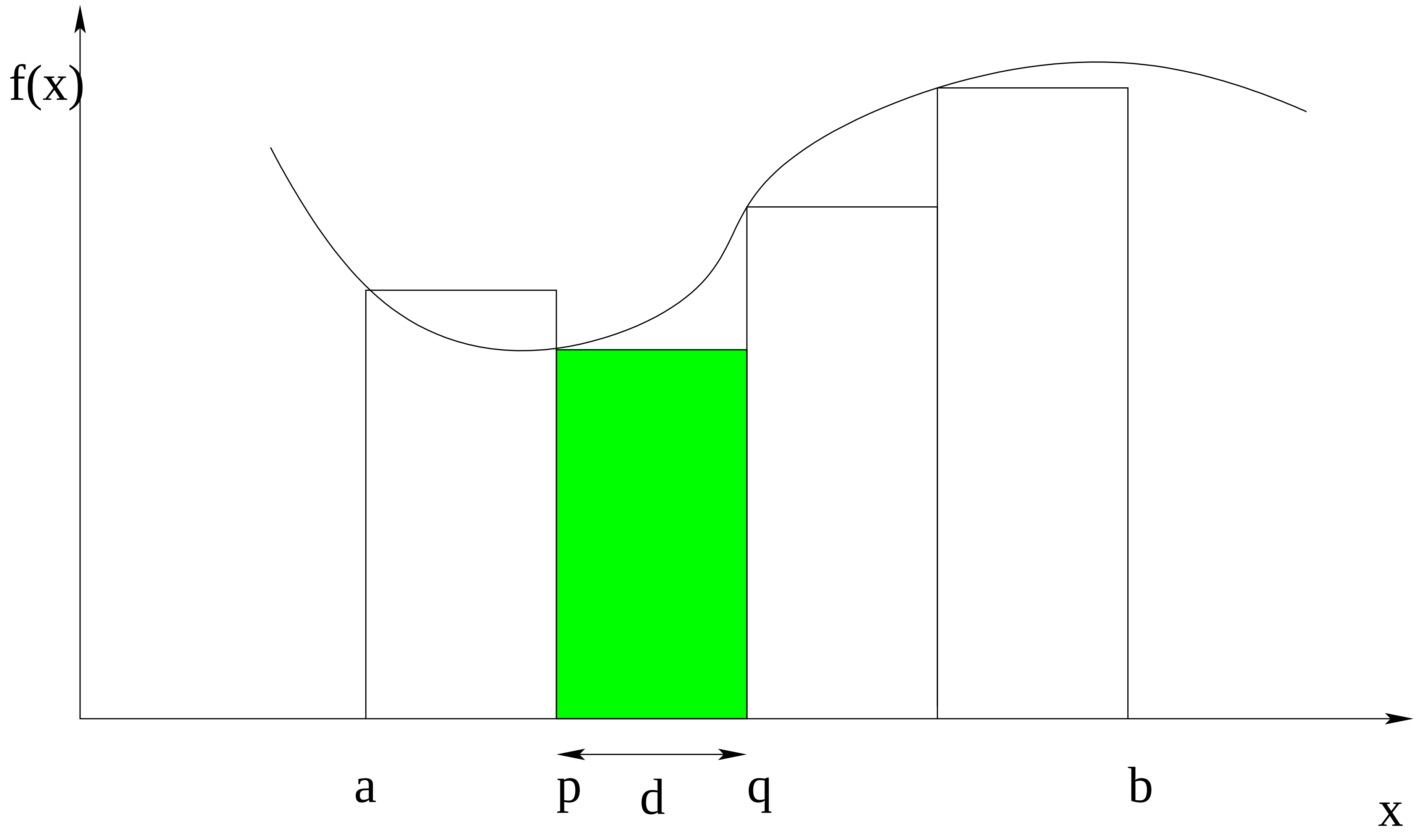
Example: For

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

divide the range $[a,b]$ into separate parts and perform the computation in each part using a separate process.

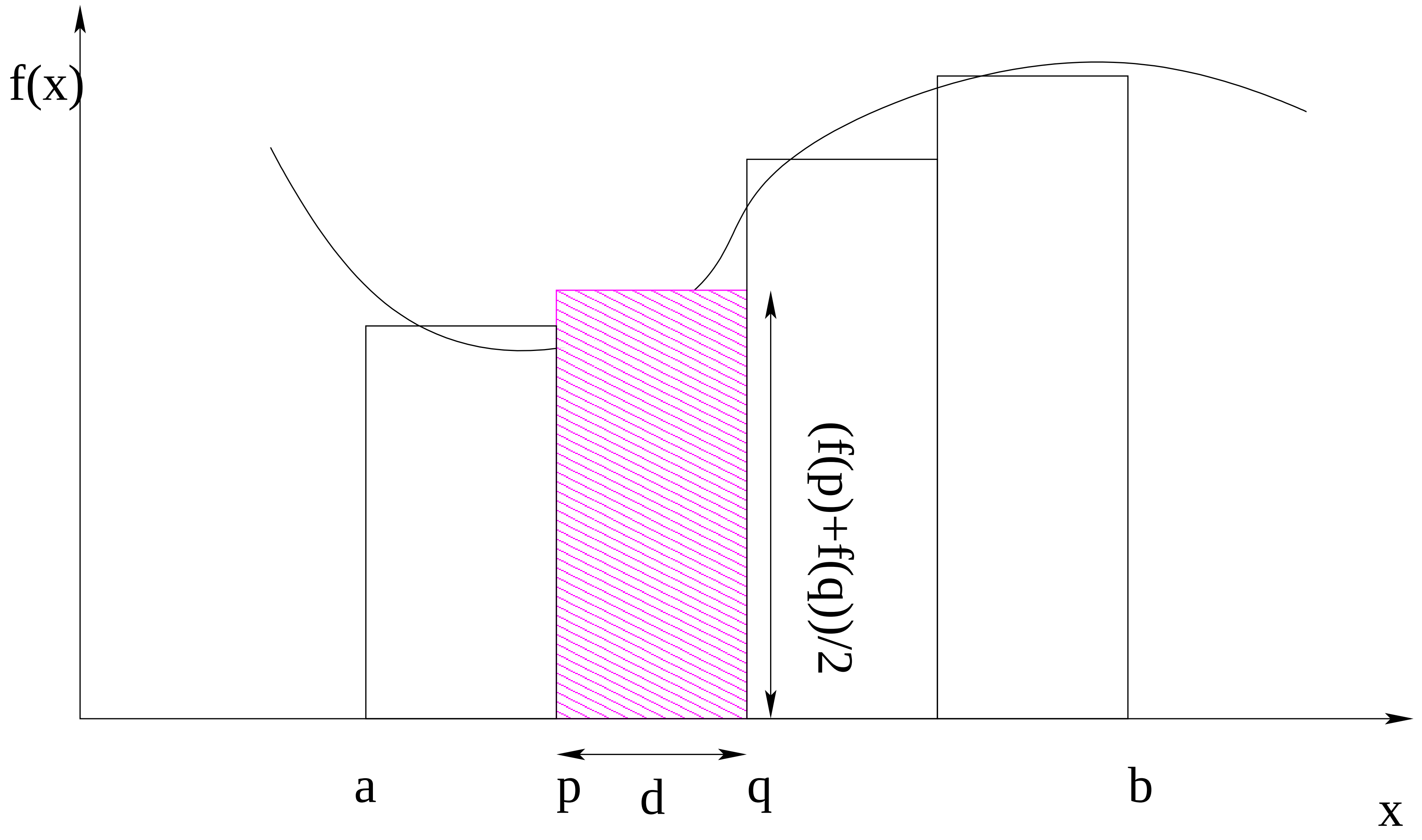
..Integration

Approximation using rectangles:



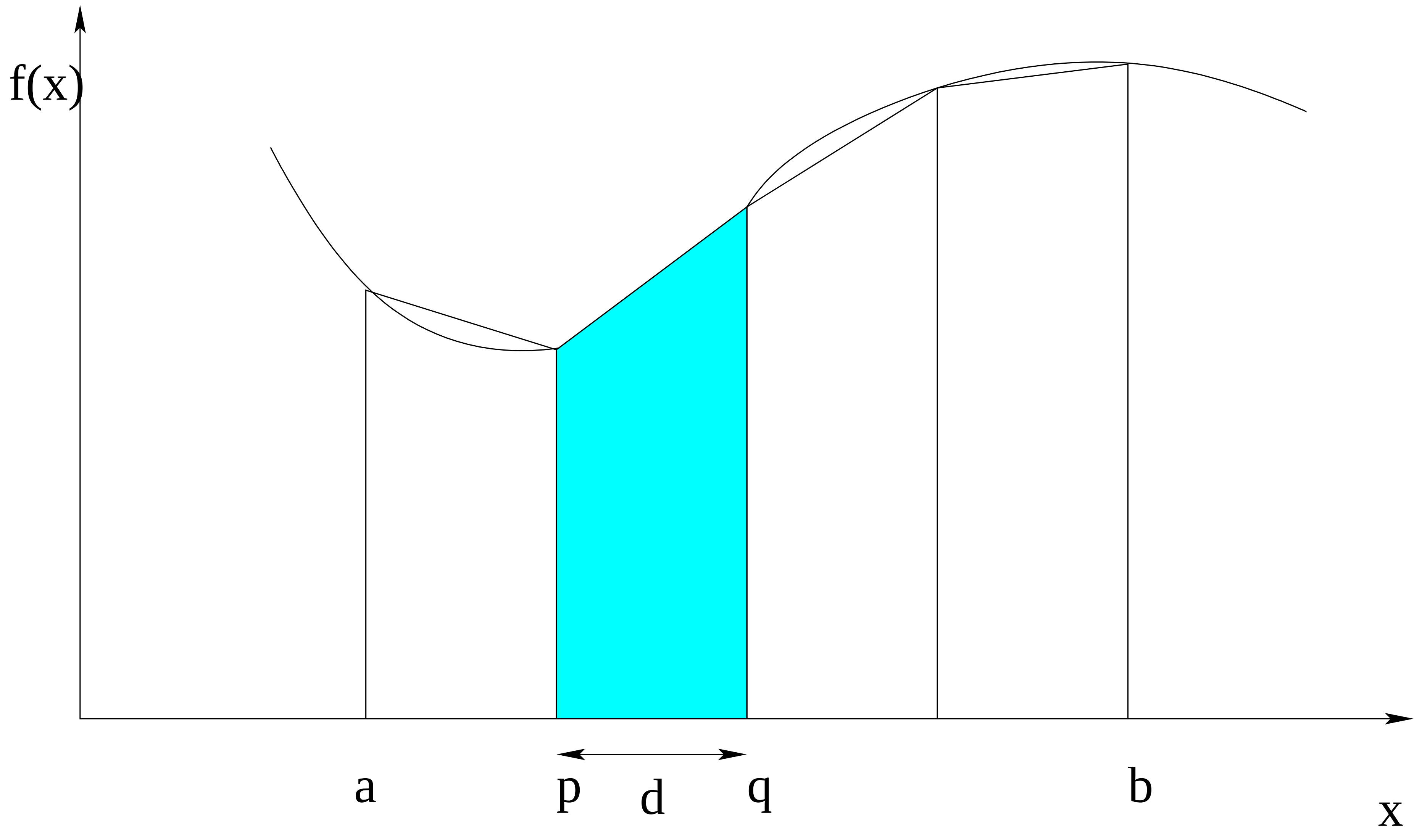
..Integration

Better approximation using rectangles (average value):



..Integration

Using trapezoidal approximations:



..Integration

Sample pseudocode:

(Static assignment, SPMD program, 2nd rectangular method)

Process P_i:

```
if (i == master){
      printf(``Enter number of interval'');
      scanf(%d,&n);
}
bcast(&n,$P_{master}$,$P_{group}$);
region = (b-a)/p;
start = a + region * i;
end = start + region;
area = 0.0;
for(x=start; x<end; x=x+d)
      area = area + f(x) + f(x+d);
area = 0.5 * area * d;
reduceAdd(&integral,&area,$P_{group}$);
```

..Integration

Final comments:

- This program is using a simple partitioning strategy. An approach using divide-and-conquer strategy may be used as well.
- The above program does not consider any termination condition. It may be (1) repeated for larger and larger n till a good approximation is obtained, or (2) the convergence criteria may be included into the program.
- Sometimes the function is not so smooth and a nonuniform division of the interval may fit better. E.g., using more processes closer to 0 gives a better approximation for $\int_{0.01}^{1}(x + sin(1/x))dx$.

Case studies: 3. Gravitational *N*-body problem

Gravitational *N*-body problem:

- This is a (computationally) difficult, practical problem which was wildly studied.
- Goal: find positions and movements of bodies in space subject to gravitational forces using Newtonian laws of physics.
- The gravitational force between two bodies of masses m_a and m_b is
$$F = \frac{Gm_am_b}{r^2}$$
where G is the gravitational constant and r is the distance between the bodies.

..*N*-body problem

- Subject to the (resulting) force F, a body will accelerate according to Newton's 2nd law
$$F = ma$$
where m is the mass of the body and a is the resulting acceleration.
- Let the time interval be Δt and v^{t+1} (resp. v^t) be the velocity at time $t+1$ (resp. t). Using the approximation
$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}$$
one gets
$$v^{t+1} = v^t + \frac{F\Delta t}{m}$$

..*N*-body problem

- The position of the body is changed from x^t to x^{t+1} by

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

- After all these computations a new position is obtained and the procedure is repeated:

  position $\mapsto$ *force* $\mapsto$ *acceleration* $\mapsto$ *velocity* $\mapsto$ *position...*

- In a 3-dim space, the distance between bodies $a = (x_a, y_a, z_a)$ and $b = (x_b, y_b, z_b)$ is

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

  and the general force gives actions along each direction

$$F_x = k(x_b - x_a)/r, \quad F_y = k(y_b - y_a)/r, \quad F_z = k(z_b - z_a)/r$$

  where $k = \frac{Gm_a m_b}{r^2}$

..*N*-body problem

Sequential (pseudo)code: (sketched, one iteration step)

```
for(t=0; t<tmax; t++)
     for(i=0; i<N; i++){
          F = Force(i); /* compute force on i-th body */
          v[i]$_{new}$ = v[i] + F * dt / m;
          x[i]$_{new}$ = x[i] + v[i]$_{new}$ * dt;
     }
for(i=0; i<N; i++){
     v[i] = v[i]$_{new}$;
     x[i] = x[i]$_{new}$;
}
```

..N-body problem

Parallel approach: (sketched)

- The algorithm uses $O(N^2)$ operations for one iteration (each body acts on each other body), hence is not practical for usual N-body problems where N is large.
- The time complexity is reduced using the observation that a cluster of distant bodies can be approximated as a single body (containing the total mass of the bodies in the cluster)

..N-body problem

Barnes-Hut algorithm:

- Start with a cube containing all bodies. The cube is divided into eight subcubes.
- If a subcube contains no particles, then the subcube is deleted from further considerations
- If a subcube contains more than one body, then it is recursively divided until every subcube contains one body.

Notice: This way octtrees *are obtained (each node has up to 8 children).*

..N-body problem

- The total mass and the center of mass is stored in each node of the tree.
- The force on each body can be obtained by traversion the tree from the root and stopping when a clustering approximation may be used, usually,

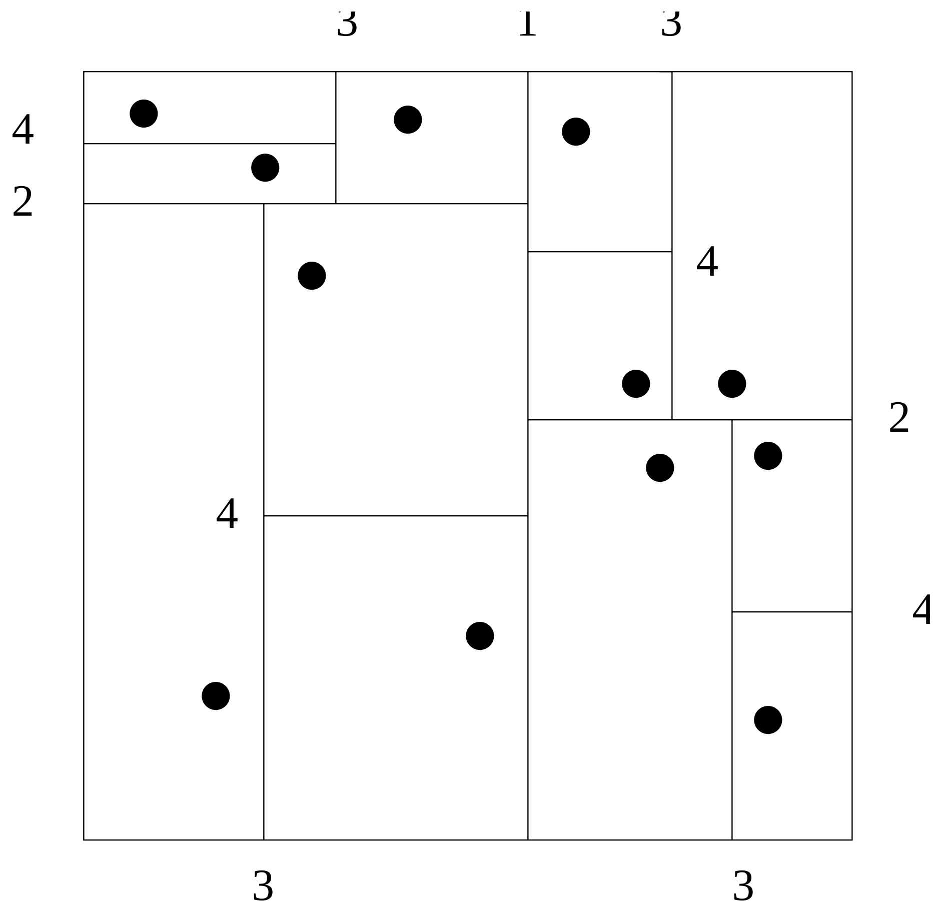
  $$r \geq d/\theta$$

  where r - distance to the mass center, d - cube dimension, and θ constant, typically 1.0.

Both, the construction of the tree and the computation of all forces require $O(n \log n)$ time, hence the algorithm is $O(n \log n)$ (rather than the direct approach requiring $O(N^2)$ time).

..*N*-body problem

The resulting tree may be not well balanced, hence a different approach will be to divide the space into subcubes having (closed to) an equal number of bodies. An illustration is below (in 2-dim case).



3
1
3
4
2
4
2
4
4
3
3