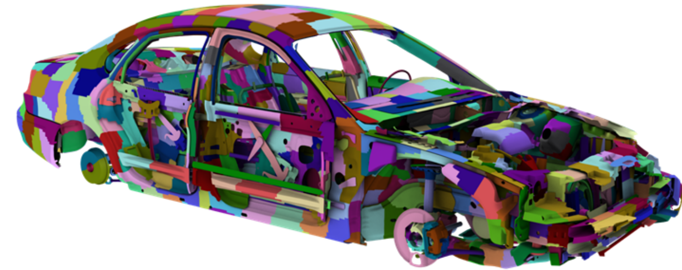
Outline: Parallelization via Partitioning and Divide-and-Conquer

- two (related) parallelization techniques: partitioning and divide-and-conquer
- example 1: addition of (centralized) 1D array entries
  - parallelization by partitioning
  - parallelization by divide-and-conquer
- example 2: numerical integration using quadratures
  - problem definition and sequential algorithm
  - parallelization by partitioning
- example 3: solving N-body problems
  - problem definition
  - all-pairs sequential algorithm
  - reducing complexity by divide-and-conquer: the Barnes-Hut algorithm
  - parallelization challenges

Ref: Wilkinson and Allen Ch 4.

Partitioning and Divide-and-Conquer

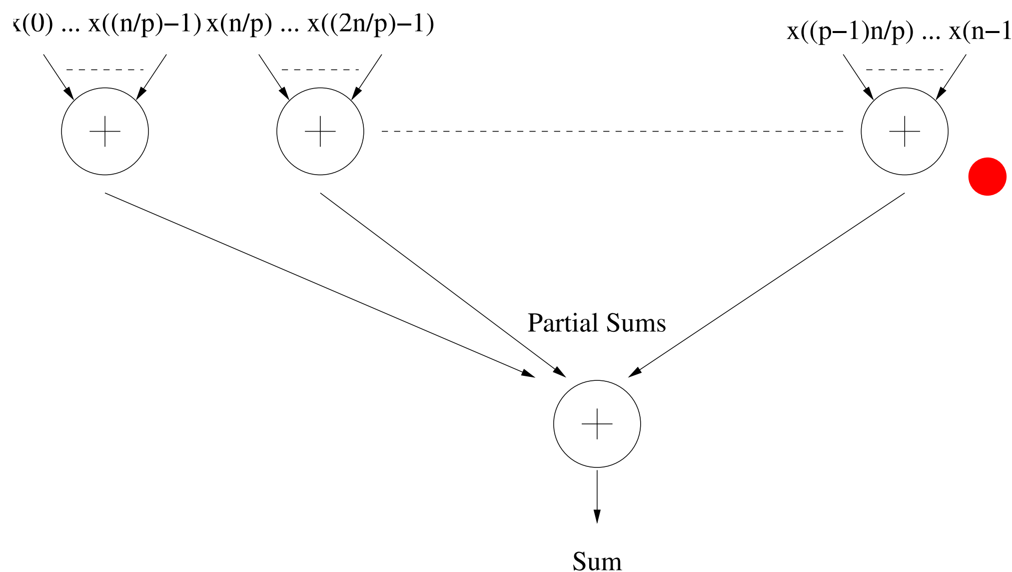
- two related parallelization approaches: partitioning and divide-and-conquer
- in partitioning, the problem is divided into separate parts, and each part is executed separately on different processors
- most partitioning formulations (except for the case of naturally parallel problems) require the results of the parts to be combined to obtain the desired result
- two variants of partitioning: data partitioning (this lecture; see figure) and functional partitioning (much less common)
- divide-and-conquer applies partitioning in a recursive manner dividing the problem into smaller and smaller tasks before solving the smaller tasks and combining the results



example: finite element meshes of complex geometries partitioned using multilevel graph partitioning algorithms

Example 1: Addition of (Centralized) 1D Array Entries

- assume we want to sum the n entries of a 1D array under the assumption that *the array is initially stored in one of the processes*, say process 0
- let us first explore the (data) partitioning approach (aka domain decomposition)
- simple strategy: divide array into p parts of size $\frac{n}{p}$ numbers each
- each part can be processed by a different process and the partial sums have to be combined (i.e., summed up) to obtain the final result (in the case of the figure, a single process is in charge of performing the final sum)



- note that each process needs the data it has to accumulate, but the array is centralized in process 0! How can we address this? (next slides)

First Approach: Master/Slave via Send/Recv to Distribute Data

Master:

```
s = n/(p-1); // size of each part
offset = 0;
for (slave=1; slave<p; slave++) {
  send(&numbers[offset], s, slave); // send chunk of s numbers to slave
  offset = offset + s;
}

sum = 0;
for (slave = 1; slave < p; slave++) {
  recv(&part_sum, 1, any_proc); // receive partial sums in any order!
  sum = sum + part_sum;          // accumulate partial sums
}
```

Slave:

```
s = n/(p-1);
master = 0
recv(numbers, s, master);    // receive chunk of s numbers from master
part_sum = 0;
for (i = 0; i < s; i++)
  part_sum = part_sum + numbers[i];
send(&part_sum, 1, master); // send partial sum to master
```

Second Approach: Master/Slave via Broadcast to Distribute Data

Master:

```
master = 0
bcast(numbers, n, master)       // broadcast the whole array to all slaves
sum = 0;
for (slave = 1; slave < p; slave++) {
  recv(&part_sum, 1, any_proc); // receive partial sums in any order!
  sum = sum + part_sum;         // accumulate partial sums
}
```

Slave:

```
master = 0;
numbers = memalloc(n);
bcast(numbers, n, master);  // get whole array from master

s = n/(p-1);
start = (me-1)*s;
end = start + s;
part_sum = 0;
for (i = start; i < end; i++)
  part_sum = part_sum + numbers[i];
send(&part_sum, 1, master); // send partial sum to master
```

The relative merit of this approach versus the previous one depends, among others, on the specifics of the broadcast implementation when unfolded in the underlying network

Third Approach: Scatter + Reduce

```
// allocate buffer space for the
// local chunk of numbers
s = n/p;
rcv_buf = memalloc(s);

// root scatters to all processes their
// respective chunks of size s
root=0;
scatter(numbers, rcv_buf, s, root);

part_sum = 0;
for (i = 0; i < s; i++)
  part_sum = part_sum + rcv_buf[i];

reduce(&part_sum, &sum, 1, root, SUM);
```

- this third approach does NOT follow the master/slave paradigm
- instead, it uses scatter + reduce (collectives) on the whole set of processes
- note that `numbers[]` array is ONLY consumed at the `root` process (it might indeed be a dangling pointer in processes different from the root)
- the final `sum` is (only) available at the root

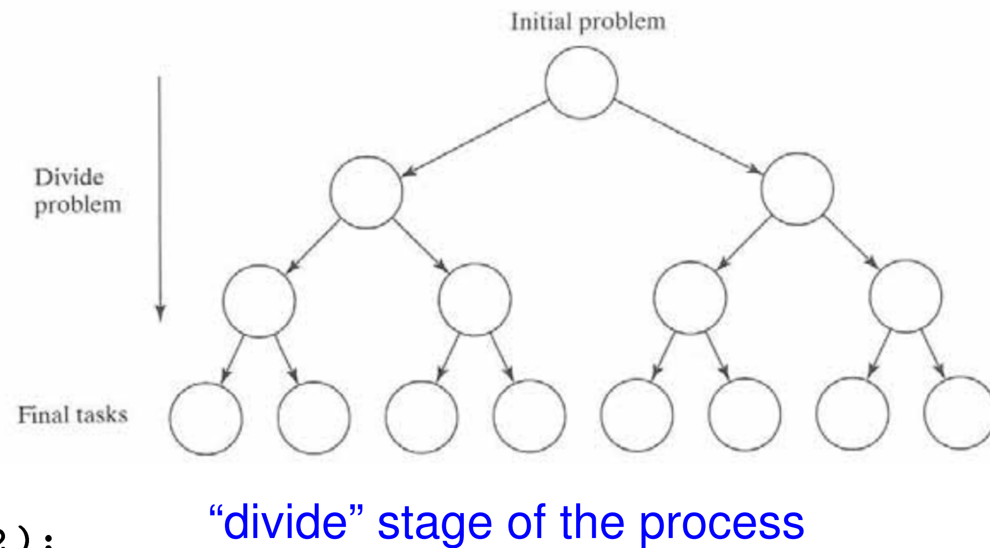
Divide-and-Conquer

- parallelization approach characterized by recursively applying partitioning to divide a large problem into smaller and smaller subproblems *of the same form as the original problem*
- recursion is a key concept for divide-and-conquer; it is applied until the tasks cannot be subdivided further or they are “small enough”
- divide-and-conquer parallel algorithms can be understood as operating in parallel with tree-like data structures
- when each subdivision creates two parts, a recursive divide-and-conquer formulation forms a binary tree (next slide), although we may have subdivision into $M > 2$ parts, leading to M-ary trees

Divide-and-Conquer Addition of 1D Array Entries (Sequential)

sequential divide-and-conquer for the addition problem may look as follows

```
// add 1D array of numbers
int add(int *s) {
  int * s1, * s2;
  if (length(s) == 0)
    return 0;
  elif (length(s) == 1)
    return (s[0]);
  else {
    // continue recursion
    divide(s, &s1, &s2);
    part_sum1 = add(s1);
    part_sum2 = add(s2);
    return (part_sum1 + part_sum2);
  }
}
```



“divide” stage of the process

- the problem is first divided into two parts
- these two parts are each divided into two parts, and so on till leaves are reached
- the basic operations (summation of entries) are performed at the leaves
- accumulation of partial results occurs bottom-top in reverse order

Divide-and-Conquer Addition of 1D Array Entries (Parallel)

- the recursive subdivision process offers opportunities for parallelism: once a subdivision is made, both parts can be processed simultaneously
- a smart (static) mapping approach of tasks to processes is the one illustrated in the figure ($p = 8$), in which we reuse processes at each tree level
- communication pattern is the same as the one of binary broadcast collective on hypercube networks (based on binary addresses)

$\log_2(p)$ stages

- stage 1: $P_0(000)$ passes 2nd half of whole array to $P_4(100)$
- stage 2: $P_0(000) \rightarrow P_2(010)$, $P_4(100) \rightarrow P_6(110)$
- stage 3: $P_0(000) \rightarrow P_1(001)$, $P_2(010) \rightarrow P_3(011)$, $P_4(100) \rightarrow P_5(101)$, $P_6(110) \rightarrow P_7(111)$
- each subarray at the leaves will have $\frac{n}{p}$ entries

Divide-and-Conquer Addition of 1D Array Entries (Parallel)

- combination of partial sums operates in reverse manner, i.e., from leaves to root
- communication pattern is the same as the one of binary reduce collective on hypercube networks (based on binary addresses)

$\log_2(p)$ stages

- stage 1: $P_1(001) \to P_0(000)$, $P_3(011) \to P_2(010)$, $P_5(101) \to P_4(100)$,
- stage 2: $P_2(010) \to P_0(000)$, $P_6(110) \to P_4(100)$
- stage 3: $P_4(100)$ sends sum of 2nd half of whole array to $P_0(000)$

Analysis of Divide-and-Conquer Centralized 1D Array Addition

- sequential algorithm time:

  $t_{\text{seq}} = (n-1)t_f$
- parallel comm 1 (divide):

  $t_{\text{comm1}} = \frac{n}{2}t_w + \frac{n}{4}t_w + \frac{n}{8}t_w + \cdots \frac{n}{p}t_w = \frac{n(p-1)}{p}t_w$
- parallel computation (divide+combine):

  $t_{\text{comp}} = (\frac{n}{p} + \log_2(p))t_f$
- parallel comm 2 (combine):

  $t_{\text{comm2}} = \log_2(p)t_w$
- parallel algorithm time:

  $t_{\text{par}} = t_{\text{comm1}} + t_{\text{comm2}} + t_{\text{comp}} = \left(\frac{n(p-1)}{p} + \log_2(p)\right)t_w + (\frac{n}{p} + \log_2(p))t_f$

Assumptions:

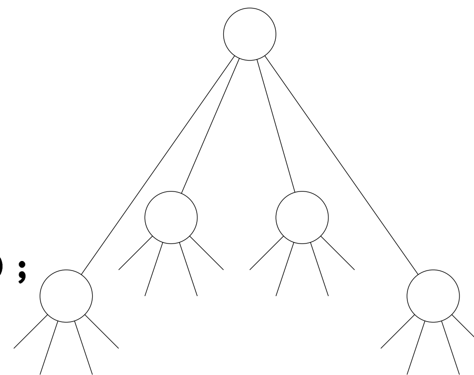
- n and p are powers of 2
- ignore message start-up time t_s
- neglect the effect of # of links and t_h
- the network provides enough parallelism to transfer messages within each stage in parallel

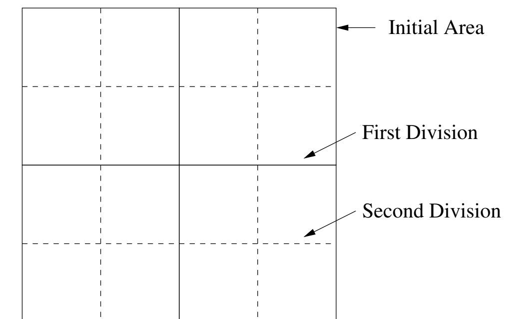
M-ary Divide-And-Conquer

- divide-and-conquer can be generalized to M-ary trees, with $M > 2$
- with $M = 4$: quadtree; with $M = 8$: octree

4-ary divide-and-conquer for the
addition problem may look as follows

```
// add 1D array of numbers
int add(int *s) {
 if ... // stop recursion
 else {
   // continue recursion
   divide(s, &s1, &s2, &s3, &s4);
   part_sum1 = add(s1);
   part_sum2 = add(s2);
   part_sum3 = add(s3);
   part_sum4 = add(s4);
   return (part_sum1 + part_sum2 +
           part_sum3 + part_sum4);
 }
}
```





use of quadtree for image processing
(2D space subdivision)

Example 2: Numerical Integration Using Quadratures (definition)

- the goal is to approximate $\int_I f(x)$ on arbitrary interval I by quadrature formulas
- for convenience, these are typically defined in a ref. interval, e.g., $\hat{K} := [-1, 1]$
- in general, any quadrature formula obeys the following general definition:

$$\int_{-1}^{1} f(x)\mathrm{d}x \approx \sum_{l=1}^{m} \omega_l f(\xi_l)$$

  where $\xi_l \in [-1, 1]$ are called quadrature nodes (and are all distinct), and the real numbers ω_l are called quadrature weights

- ξ_l and ω_l are judiciously chosen such that approximation fulfills certain properties, *typically that polynomials up to a certain degree can be integrated exactly*
- by a change of variables, we can transform the formula above on any interval $[a, b]$:

$$\int_{a}^{b} f(x)\mathrm{d}x \approx \sum_{l=1}^{m} \frac{1}{2}\delta\omega_l f(c + \frac{1}{2}\delta\xi_l)$$

  where $c := \frac{1}{2}(a+b)$ and $\delta := b - a$

Numerical Integration Using Quadratures (trapezoidal rule)

- among all possible quadrature formulas, we are going to use the trapezoidal rule
- this rule is defined as $m = 2$, $\xi_1 = -1$, $\xi_2 = 1$, $\omega_1 = \omega_2 = 1$
- if we replace these values in the above formula, then we obtain:

$$\int_a^b f(x)\mathrm{d}x \approx \frac{1}{2}(b-a)(f(a)+f(b))$$

- the trapezoidal is able to integrate exactly polynomials of up to degree 1 (i.e., linear functions)
- it actually belongs to a general class of quadratures referred to as Gauss-Lobatto-Legendre quadratures
- how can we reduce the approximation error for more complicated functions (e.g., trigonometric functions)? (next slide)

Numerical Integration Using Quadratures (sequential algorithm)

- we split I into n fixed-size subintervals (note there are also variants which built varying-size intervals dynamically/recursively using some optimization criterion)
- thus, definite integral over I decomposed as sum of integrals over each subinterval
- we use trapezoidal rule to approximate integrals over subintervals
- the larger the n, the better the approximation

```
printf("Enter the number
        subintervals\n");
scanf("%d",&n);
d = (b-a)/n;
area = 0.0;
x = a;
for (i = 0; i < n; i++) {
  area = area + f(x) + f(x+d);
  x=x+d;
}
area=0.5*area*d
```


$n = 4$

Numerical Integration Using Quadratures (parallel algorithm)

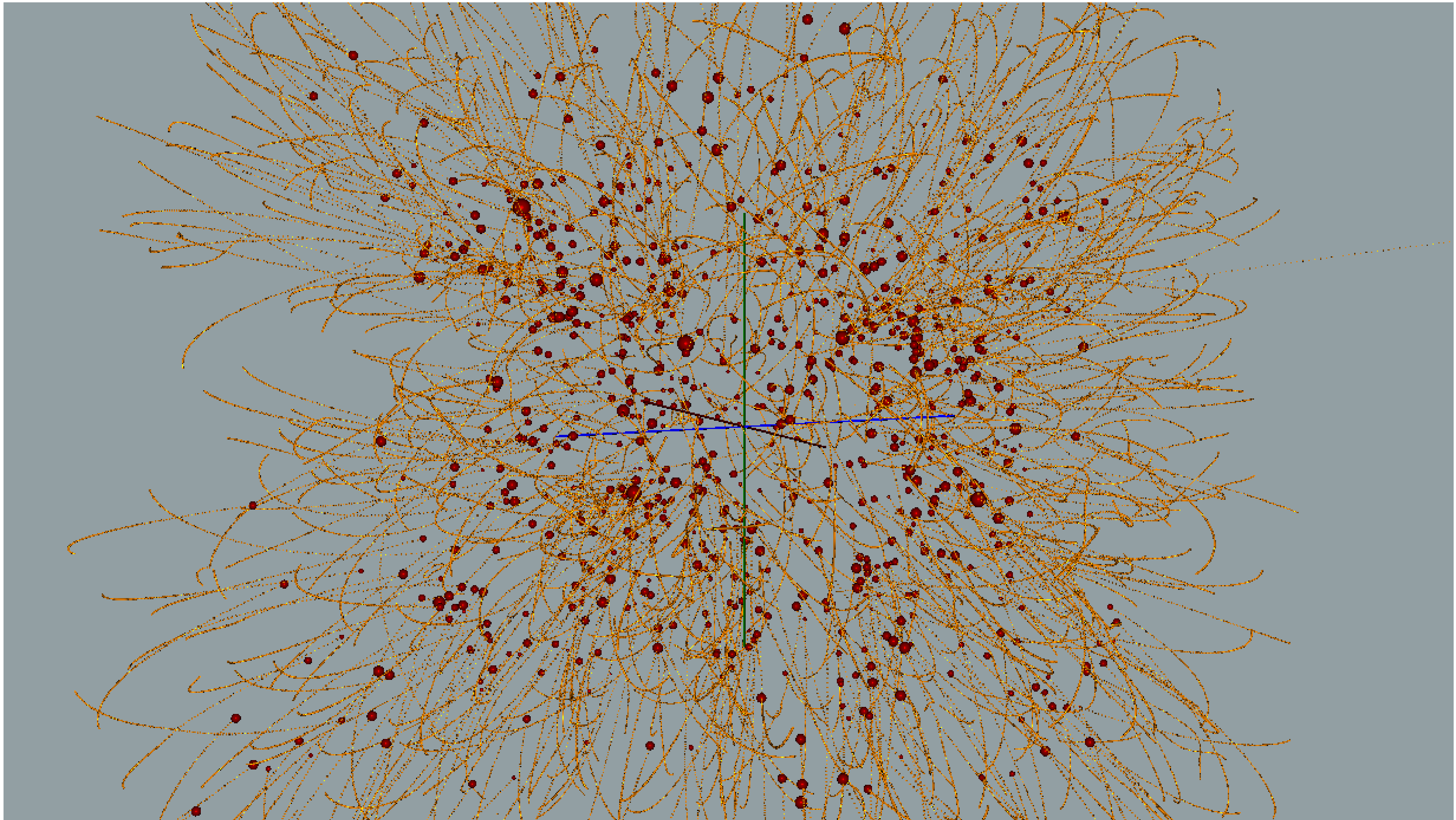
```
me=process_rank_id();
p=num_ranks();
root=0;

if (me == root) {
  printf("Enter number of
          subintervals\n");
  scanf("%d",&n);
}
bcast(&n, 1, root);
region_length = (b-a)/p;
x = a + me*region_length;
d = (b-a)/n;
part_area = 0.0;
for (i = 0; i < n/p; i++) {
  part_area += f(x) + f(x+d);
  x = x + d;
}
reduce(&part_area, &area, 1, root, SUM);
if (me == root) {
  area=0.5*area*d;
  printf("Area under the
          curve is %e\n", area);
}
```

- data partitioning parallelization
- each process approximates the integral over a different region within the $[a,b]$ interval
- partial sums are finally combined (i.e., summed up using) reduce

Example #3: Solving N-body Problems (problem definition)

- a general class of problems which aim at describing the dynamics (instantaneous positions/velocities) of N particles under forces which these exert on each other
- for conciseness, we restrict to gravitational N-body problems, i.e., the forces among the particles are determined by Newton's law of universal gravitation
- however, the same concept applies to other fields, e.g., electrically charged particles, molecular dynamics, fluid dynamics, etc.
- N-body problems are mathematically modelled by systems of differential equations. However, analytical solutions for these are not easy to determine (if possible) for $N > 3$, so that computational methods have to be used
- N-body problems computer simulations can greatly benefit from divide-and-conquer (e.g. recursive space subdivision in the Barnes-Hut algorithm)

N-body Simulation Snapshot



Source: https://philippos.info/nbody/

Newton's Law of Universal Gravitation (refresher)

- assume we have two bodies identified as i, j, with $i, j = 1 \ldots N$, masses m_i and m_j resp., and position vectors $\mathbf{x}_i$ and $\mathbf{x}_j$, resp., then the gravitational force that j exerts on i is given by:

$$\mathbf{F}_{ij} = G \frac{m_i m_j}{||\mathbf{x}_{ij}||^2} \frac{\mathbf{x}_{ij}}{||\mathbf{x}_{ij}||}$$

  where G is the gravitational constant, and $\mathbf{x}_{ij} = \mathbf{x}_j - \mathbf{x}_i$

- thus, if we know the mass of the bodies, and their position vectors, then we can compute the resultant of forces on each body, i.e., $\mathbf{F}_i = \sum_{j \neq i} \mathbf{F}_{ij}$
- given $\mathbf{F}_i$ and m_i, we can compute the acceleration (instantaneous change in velocity) caused by the resultant force on each body $i = 1, \ldots, N$, using Newton's second law

Discretization; Updating Approximate Velocities and Positions

- as usual, we discretize (split) time simulation interval $[0,T]$ into M subintervals of length δt
- our aim is to approximate the position vectors and velocities at the start of each time subinterval
- let us denote by $\mathbf{v}_i^n$ and $\mathbf{x}_i^n$ the approximate velocity and position vector of the i-th body at the start of subinterval n
- assuming that the force $\mathbf{F}_i$ (and thus acceleration) is constant through the whole time subinterval n (*clearly not true!*), then $\mathbf{v}_i^{n+1}$ approximated by

$$\mathbf{v}_i^{n+1} = \mathbf{v}_i^n + \frac{\mathbf{F}_i}{m_i}\delta t$$

- analogously, the position vector can be approximated as:

$$\mathbf{x}_i^{n+1} = \mathbf{x}_i^n + \mathbf{v}_i^n \delta t$$

- for "small enough" δt, discretization gives a reasonable approximation to the actual physical phenomena

Sequential Algorithm (All-Pairs)

```
dt = T/M
for (t=0;t<M;t++)                    // loop over time subintervals
{
  for (i=0;i<N;i++) {                // loop over N bodies
    Initialize Fi to zero
    for (j=0;j<N;j++) {              // loop over N bodies
      compute Fij                    // force that j exerts on i
      Fi += Fij
    }
    vi_new = vi + Fi*dt/mi           // update velocity
    xi_new = xi + vi*dt              // update position
  }
  for (i=0;i<N;i++)                  // loop over N bodies
  {
    vi = vi_new
    xi = xi_new
  }
}
```

- known as “all-pairs” algorithm in the literature
- number of operations can be cut in half by only calculating $\mathbf{F}_{ij}$, for $i = 1,\dots N$ and $j = 1,\dots i-1$ (note that $\mathbf{F}_{ij} = -\mathbf{F}_{ji}$)
- in any case, $O(N^2)$ complexity!
- can we do better? (i.e., reduce the order of complexity)

Barnes-Hut Algorithm (Overview)

- proposed by Barnes & Hut in 1986
- basic idea: clustering (see figure)
- builds a quadtree (2D) or octtree (3D) by recursive space subdivision (i.e., divide-and-conquer)
- tree traversals are used to both compute masses/centre of masses of clusters and forces among particles and clusters
- reduces complexity from $O(N^2)$ to $O(N \log(N))$

Center of Mass

r

Distant Cluster of Bodies

the forces exerted by several bodies that are clustered together but are located at large r from another body can be approximated by the force exerted by a clustered body located at the center of mass of the cluster

Barnes-Hut Algorithm (Tree Construction in 2D)

- assumes a 2D space with fixed boundaries and embeds it within one square
- if more than one body, divide the square into 4 subsquares
- subsquares with more than 1 body are recursively divided into 4 again, while subsquares with no bodies in them are not subdivided by tagged as void
- continue until all leaves have only one particle (or none)
- *the tree is used to cluster the bodies together* when computing the forces (we have as many potential clusters as intermediate nodes in the tree)

Barnes-Hut Algorithm (Computation of Forces)

masses and centre of masses of (potential) clusters:

for each level ℓ, from fine to coarse:
    set to zero the masses of all void cells on level ℓ
    for each non-void cell c on level ℓ:
        if c has children:
          compute the total mass and center of mass
          for cell c by considering its children
        else:
          set the total mass and center of mass
          for cell c to the mass and position of only body

Spatial Domain

Quad-Tree Representation

body in red is "far enough away" from cluster in green if $\frac{L}{D} \leq \theta$, with $\theta < 1$ user-prescribed parameter

actual computation of forces:

for each particle p:
    for each cell c on the top level
        if c is "far enough away" (see fig) from p:
            use the total mass and center of mass of c;
        otherwise consider the children of c

Barnes-Hut Algorithm (Parallelization Challenges)

- message-passing parallelization of Barnes-Hut Algorithm is challenging
- first, *the problem is irregular*, the bodies might not distributed uniformly across space, challenging load balancing
- second, the *irregularity dynamically varies in time* as the bodies interact with each other, so that dynamic load rebalancing is in general needed to keep the number of bodies per process balanced
- third, for scalable parallelization, all stages have to be parallelized and the tree has to be partitioned/distributed into the different processes (such that no single process can hold the whole tree)
- due to scope/time constraints, this course does not cover how to tackle all these challenges. However, the interested (and intrepid) reader might find a detailed parallelization approach in this very nice paper:

  Scalable parallel formulations of the Barnes–Hut method for n-body simulations, *Parallel Computing*, 23(5-6), pp. 797-822, 1998. Publisher: Elsevier.