Cosmology Applications
N-Body Simulations

Credits: Lecture Slides of Dr. James Demmel, Dr. Kathy Yelick, University of California, Berkeley

Introduction

- Classical N-body problem simulates the evolution of a system of N bodies
- The force exerted on each body arises due to its interaction with all the other bodies in the system

Motivation

- Particle methods are used for a variety of applications
- Astrophysics
  - The particles are stars or galaxies
  - The force is gravity
- Particle physics
  - The particles are ions, electrons, etc.
  - The force is due to Coulomb's Law
- Molecular dynamics
  - The particles are atoms or
  - The forces is electrostatic
- Vortex methods in fluid dynamics
  - Particles are blobs of fluid

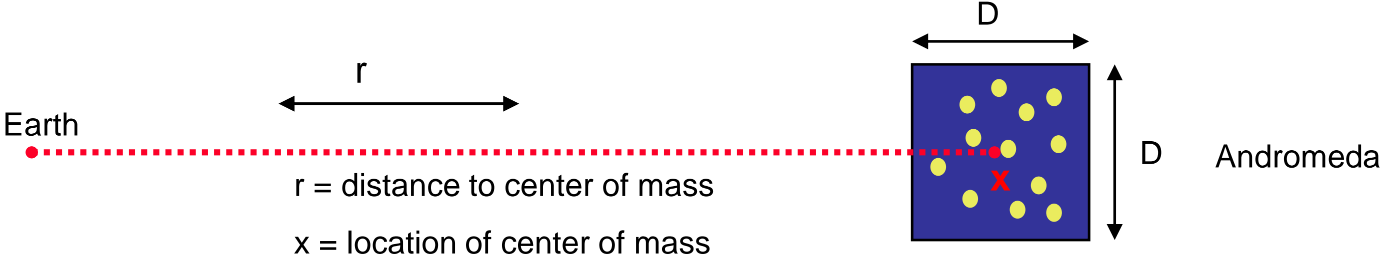
Particle Simulation

```
t = 0
while t < t_final
    for i = 1 to n                    … n = number of particles
        compute f(i) = force on particle i
    for i = 1 to n
        move particle i under force f(i) for time dt    … using F=ma
    compute interesting properties of particles (energy, etc.)
    t = t + dt
end while
```

- N-Body force (gravity or electrostatics) requires all-to-all interactions
  - $f(i) = \Sigma_{k!=i} f(i,k)$ … $f(i,k)$ = force on i from k
  - Obvious algorithm costs $O(N^2)$, but we can do better...

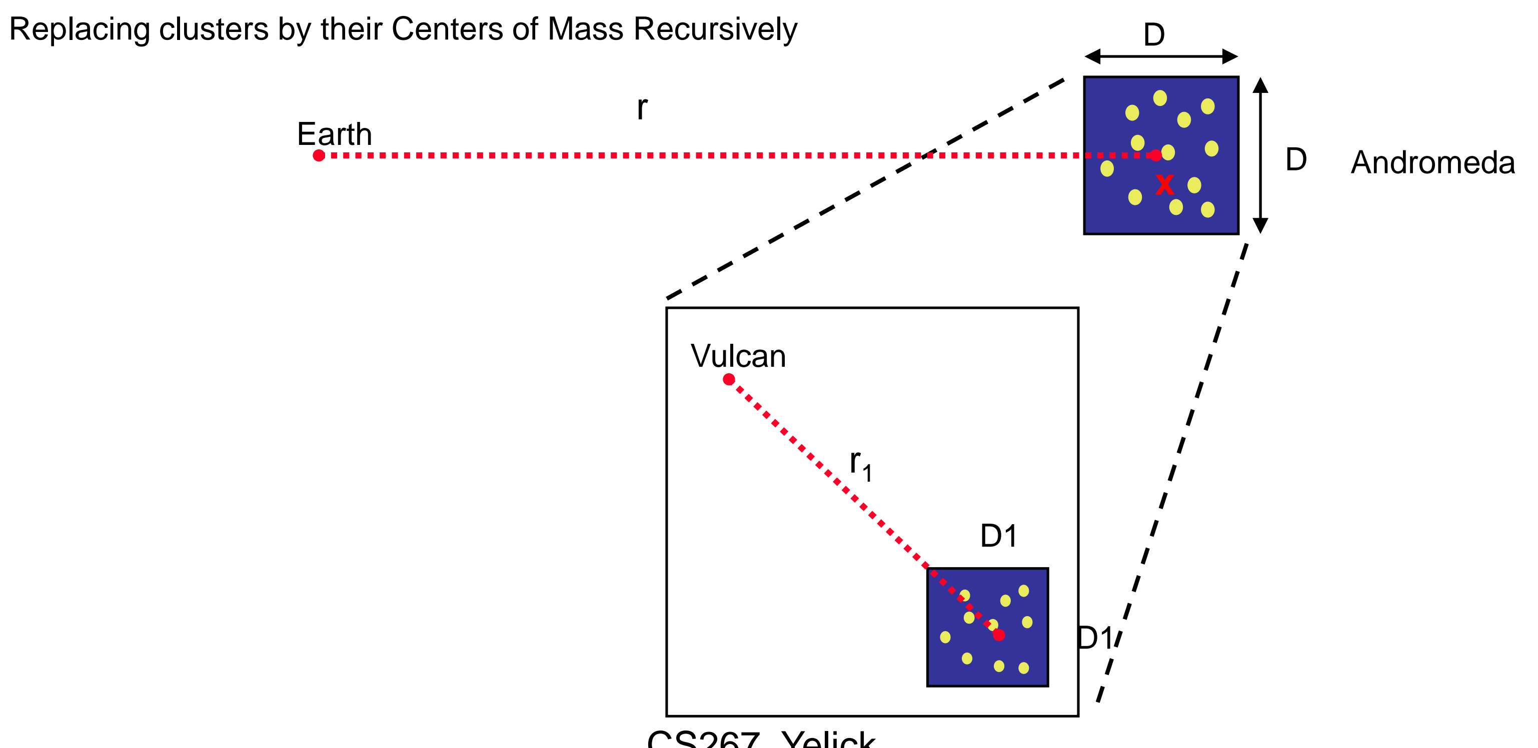
Reducing the Number of Particles in the Sum

- Consider computing force on earth due to all celestial bodies
  - Look at night sky, # terms in force sum >= number of visible stars
  - One "star" is really the Andromeda galaxy, which is billions of stars
    - A lot of work if we compute this per star …
- OK to approximate all stars in Andromeda by a single point at its center of mass (CM) with same total mass
  - D = size of box containing Andromeda , r = distance of CM to Earth
  - Require that D/r be "small enough"



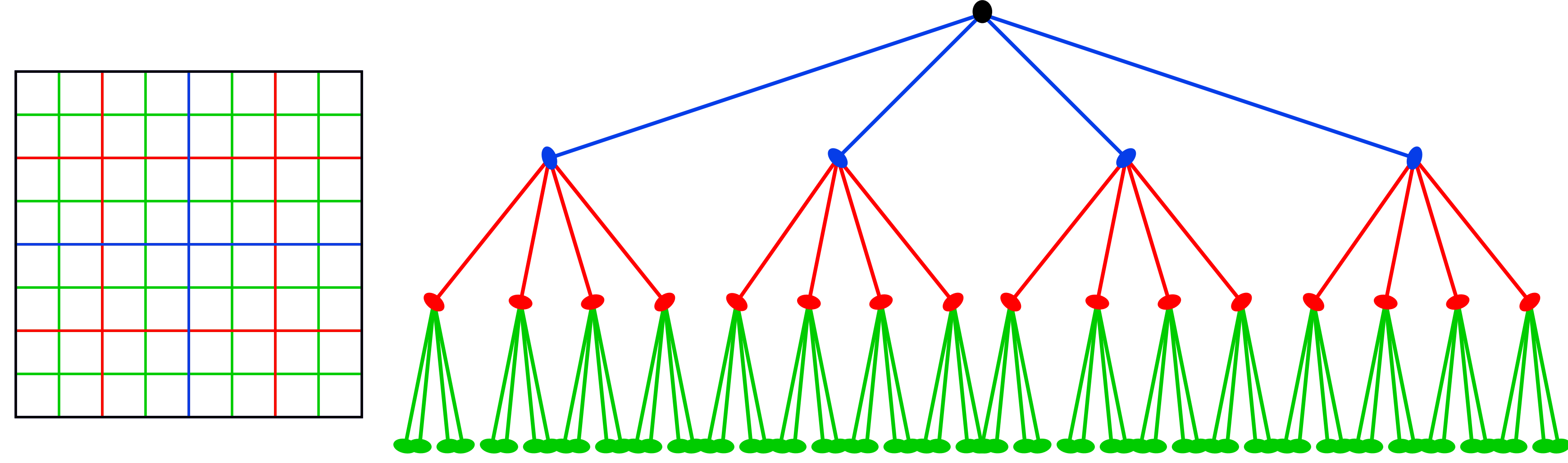
Using points at CM *Recursively*

- From Andromeda's point of view, Milky Way is also a point mass
- Within Andromeda, picture repeats itself
  - As long as D1/r1 is small enough, stars inside smaller box can be replaced by their CM to compute the force on Vulcan
  - Boxes nest in boxes recursively



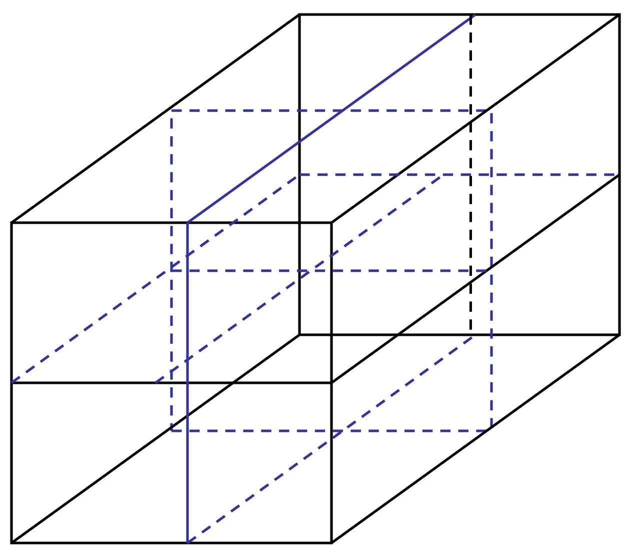
Quad Trees

- Data structure to subdivide the plane
  - Nodes can contain coordinates of center of box, side length
  - Eventually also coordinates of CM, total mass, etc.
- In a complete quad tree, each nonleaf node has 4 children

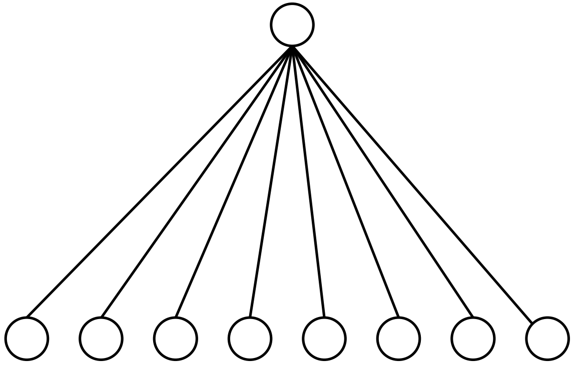


Oct Trees

- Similar Data Structure to subdivide 3D space
- Analogous to 2D Quad tree--each cube is divided into 8 sub-cubes



Two Levels of an OctTree

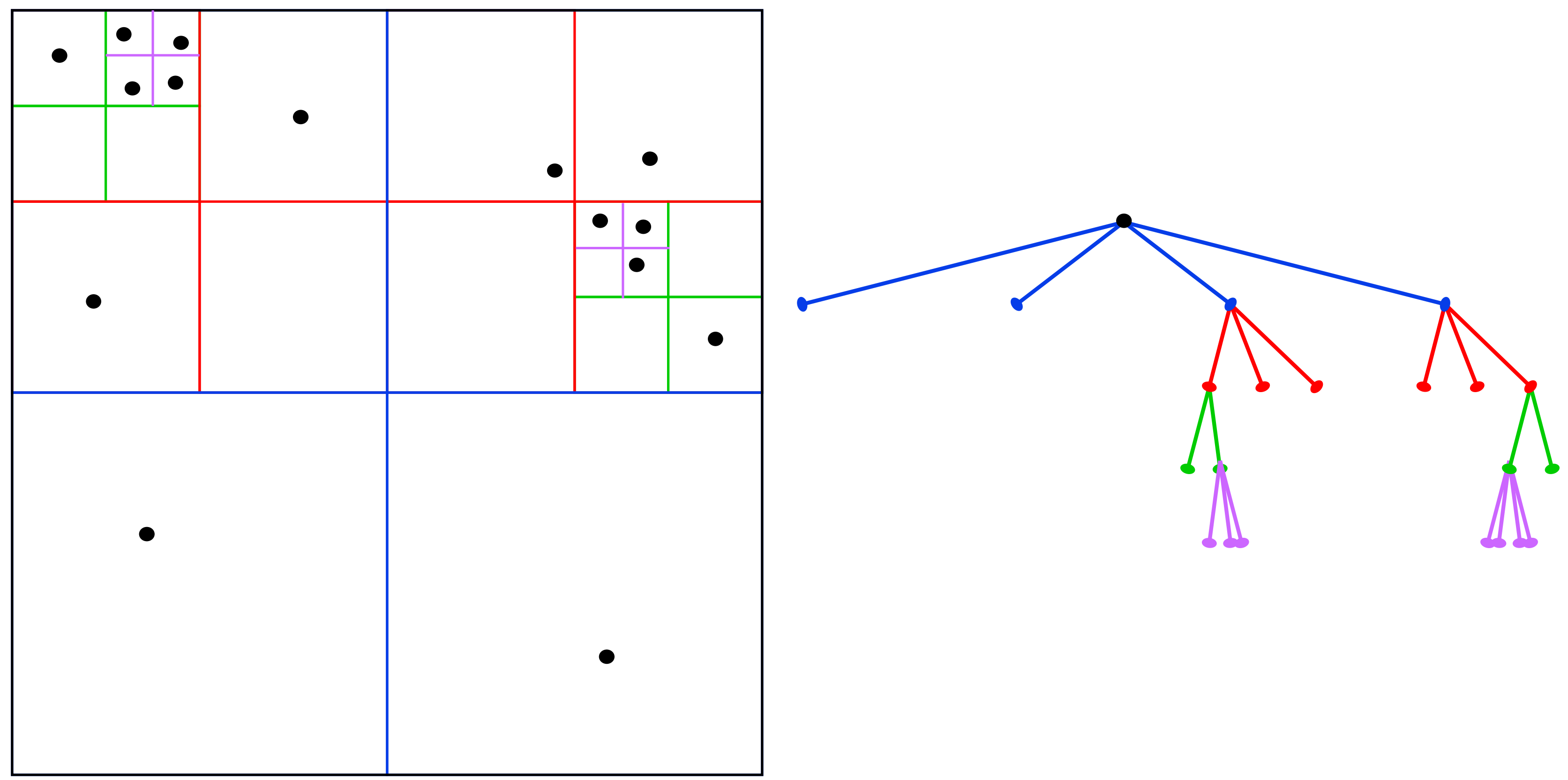


Using Quad Trees and Oct Trees

- All our algorithms begin by constructing a tree to hold all the particles
- Interesting cases have non-uniform particle distribution
  - In a complete tree (full at lowest level), most nodes would be empty, a waste of space and time
- Adaptive Quad (Oct) Tree only subdivides space where particles are located
  - More compact and efficient computationally, but harder to program

Example of an Adaptive Quad Tree

Adaptive quad tree where no space contains more than 1 particle



Child nodes enumerated counterclockwise from SW corner
Empty ones excluded

Adaptive Quad Tree Algorithm (Oct Tree analogous)

```
Procedure QuadTreeBuild
  QuadTree = {emtpy}
  for j = 1 to N                     … loop over all N particles
     QuadTreeInsert(j, root)         … insert particle j in QuadTree
  endfor
  …  At this point, the QuadTree may have some empty leaves,
  …  whose siblings are not empty
  Traverse the QuadTree eliminating empty leaves  … via, say Breadth First Search
```

```
Procedure QuadTreeInsert(j, n)
  … Try to insert particle j at node n in QuadTree
  … By construction, each leaf will contain either 1 or 0 particles
  if the subtree rooted at n contains more than 1 particle   … n is an internal node
     determine which child c of node n contains particle j
     QuadTreeInsert(j, c)
  else if the subtree rooted at  n contains 1 particle   …  n is a leaf
     add n's 4 children to the QuadTree
     move the particle already in n into the child containing it
     let c be the child of n containing j
     QuadTreeInsert(j, c)
  else  … the subtree rooted at n is an empty leaf
     store particle j in node n
  end
```

- Cost <= N * maximum cost of QuadTreeInsert = O(N * maximum depth of QuadTree)
- Uniform distribution => depth of QuadTree = O(log N), so Cost = O(N log N)
- Arbitrary distribution => depth of Quad Tree = O(b) = O(# bits in particle coords), so Cost = O(bN)

Barnes-Hut Algorithm

- High Level Algorithm (in 2D, for simplicity)

1) Build the QuadTree using QuadTree.build
… already described, cost = O(N log N)
2) For each node = subsquare in the QuadTree, compute the CM and total mass (TM) of all the particles it contains
… "post order traversal" of QuadTree, cost = O(N log N)
3) For each particle, traverse the QuadTree to compute the force on it, using the CM and TM of "distant" subsquares
… core of algorithm
… cost depends on accuracy desired but still O(N log N)

Step 2 of BH: compute CM and total mass of each node

```
… Compute the CM = Center of Mass and TM = Total Mass of all the particles
… in each node of the QuadTree
( TM, CM ) = Compute_Mass( root )
```

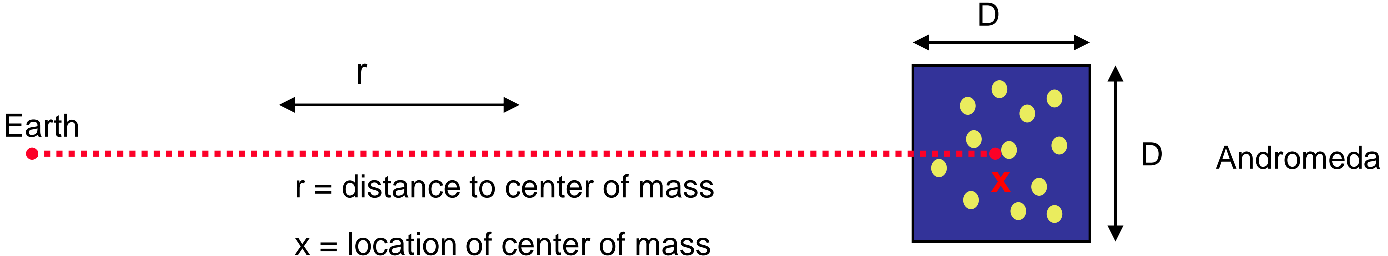
```
function ( TM, CM ) = Compute_Mass( n )    … compute the CM and TM of node n
    if n contains 1 particle
        … the TM and CM are identical to the particle's mass and location
        store (TM, CM) at n
        return (TM, CM)
    else      … "post order traversal": process parent after all children
        for all children c(j) of n  … j = 1,2,3,4
            ( TM(j), CM(j) ) = Compute_Mass( c(j) )
        endfor
        TM = TM(1) + TM(2) + TM(3) + TM(4)
             … the total mass is the sum of the children's masses
        CM = ( TM(1)*CM(1) + TM(2)*CM(2) + TM(3)*CM(3) + TM(4)*CM(4) ) / TM
             … the CM is the mass-weighted sum of the children's centers of mass
        store ( TM, CM ) at n
        return ( TM, CM )
    end if
```

Cost = O(# nodes in QuadTree)

= O(N log N) or O(b N)

Step 3: Compute Force on Each Particle

- For each node, can approximate force on particles outside the node due to particles inside node by using the node's CM and TM
- This will be accurate enough if the node if "far enough away" from the particle
- Need criterion to decide if a node is far enough from a particle
  - D = side length of node
  - r = distance from particle to CM of node
  - θ = user supplied error tolerance < 1
  - Use CM and TM to approximate force of node on box if $D/r < \theta$



Computing Force on a Particle Due to a Node

- Use example of Gravity ($1/r^2$)
- Given node n and particle k, satisfying $D/r < \theta$
  - Let (x_k, y_k, z_k) be coordinates of k, m its mass
  - Let (x_{CM}, y_{CM}, z_{CM}) be coordinates of CM
  - $r = ((x_k - x_{CM})^2 + (y_k - y_{CM})^2 + (z_k - z_{CM})^2)^{1/2}$
- G = gravitational constant
- Force on k ~

$$G * m * TM * \frac{x_{CM} - x_k}{r^3} \quad \frac{y_{CM} - y_k}{r^3} \quad \frac{z_{CM} - z_k}{r^3}$$

Details of Step 3 of BH

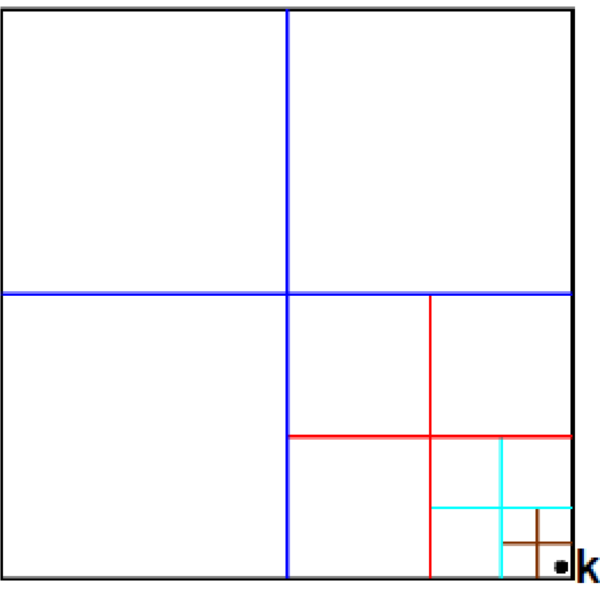
```
… for each particle, traverse the QuadTree to compute the force on it
for k = 1 to N
   f(k) = TreeForce( k, root )
               … compute force on particle k due to all particles inside root
endfor
```

```
function f = TreeForce( k, n )
   … compute force on particle k due to all particles inside node n
   f = 0
   if n contains one particle  … evaluate directly
      f = force computed using formula on last slide
   else
      r = distance from particle k to CM of particles in n
      D = size of n
      if  D/r  < θ     … ok to approximate by CM and TM
          compute f using formula from last slide
      else             … need to look inside node
          for all children c of n
              f = f + TreeForce ( k, c )
          end for
      end if
   end if
```

Analysis of Step 3 of BH

- **Correctness follows from recursive accumulation of force from each subtree**
  - Each particle is accounted for exactly once, whether it is in a leaf or other node
- **Complexity analysis**
  - Cost of TreeForce(k, root) = O(depth in QuadTree of leaf containing k)
  - Proof by Example (for $\theta > 1$):
    - For each undivided node = square, (except one containing k), $D/r < 1 < \theta$
    - There are 3 nodes at each level of the QuadTree
    - There is O(1) work per node
    - Cost = O(level of k)
  - Total cost = $O(\Sigma_k$ level of k) = O(N log N)
    - Strongly depends on θ

Sample Barnes-Hut Force calculation
For particle in lower right corner
Assuming theta > 1



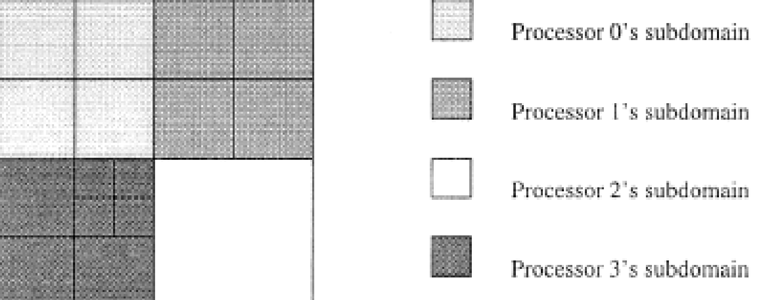
Parallelization

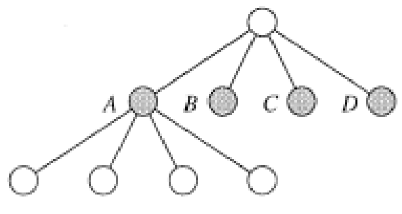
- Three phases in a single time-step: tree construction, tree traversal (or force computation), and particle advance
- Each of these must be performed in parallel; a tree cannot be stored at a single processor due to memory limitations
- Step 1: Tree construction - Processors cooperate to construct partial image of the entire tree in each processor

Step 1: Tree Construction

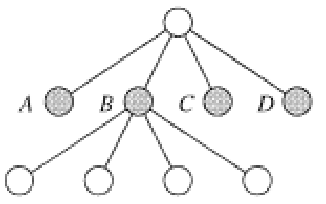
- Initially, the particles are distributed to processors such that all particles corresponding to a subtree of hierarchical domain decomposition are assigned to a single processor
- Each processor can independently construct its tree
- The nodes representing the processor domains at the coarsest level (branch nodes) are communicated to all processors using all-to-all
- Using these branch nodes, the processor reconstructs the top parts of the tree independently
- There is some amount of replication of the tree across processors since top nodes in the tree are repeatedly accessed

Step 1: Illustration

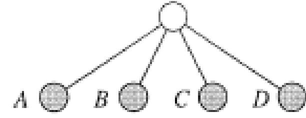




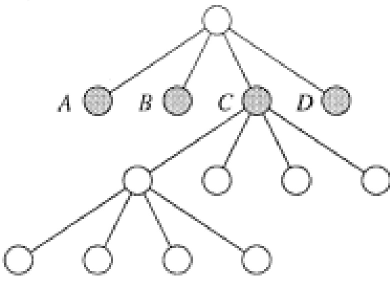
Processor 0's tree



Processor 1's tree



Processor 2's tree



Processor 3's tree

Branch nodes

Step 2: Force Computation

- To compute the force on a particle (belonging to processor A) by a node (belonging to processor B), processors need to communicate if the particle and the node belong to different processors
- Two methods:

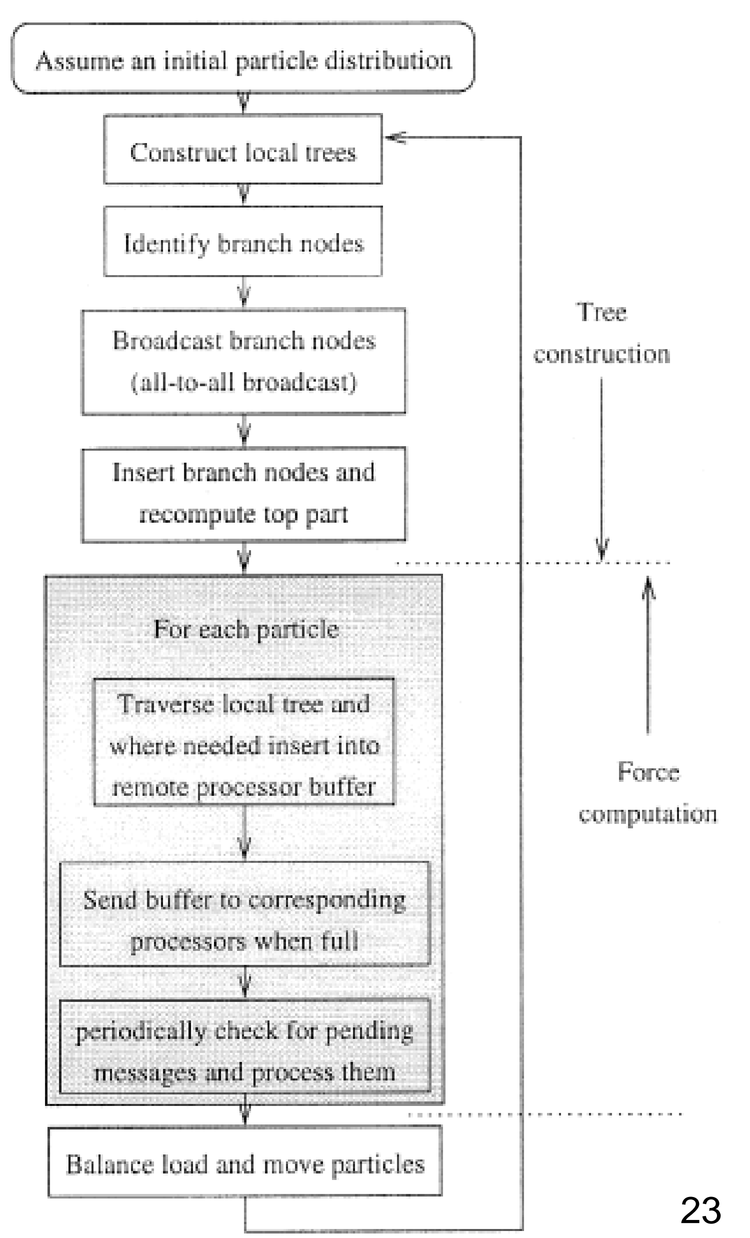
1. Children of nodes of another processor (proc B) are brought to the processor containing the particle (proc A) for which the force has to be computed
    1. Also called as *data-shipping* paradigm
    2. Follows *owner-computes* rule

Step 2: Force computation

- Method 2: Alternatively, the owning processor (proc A) can ship the particle coordinates to the other processor (proc B) containing the subtree
  1. Proc B then computes the contribution of the entire subtree on particle
  2. Sends the computed potential back to proc A
  3. *Function-shipping* paradigm: computation (or function) is shipped to the processor holding the data
  4. Communication volume is less when compared to data-shipping

Step 2: Force Computation

- In function-shipping, it is desirable to send many particle coordinates together to amortize start-up latency:
    - A processor keeps storing its particle coordinates to bins maintained for each of the other processor
    - Once a bin reaches a capacity, it is sent to the corresponding processor
- Processors must periodically process remote work requests



Load Balancing

- In applications like astrophysical simulations, high energy physics etc., the particle distributions across the domain can be highly irregular; hence tree may be very imbalanced
- Method 1: Static partitioning, static assignment (SPSA)
- Partition the domain into r subdomains, r >> p processors
- Assign r/p subdomains to each processor
- Some measure of load balance if r is sufficiently large

(a)

(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)
(0,0)	(0,1)	(0,0)	(0,1)
(1,0)	(1,1)	(1,0)	(1,1)

(b)

Load Balancing

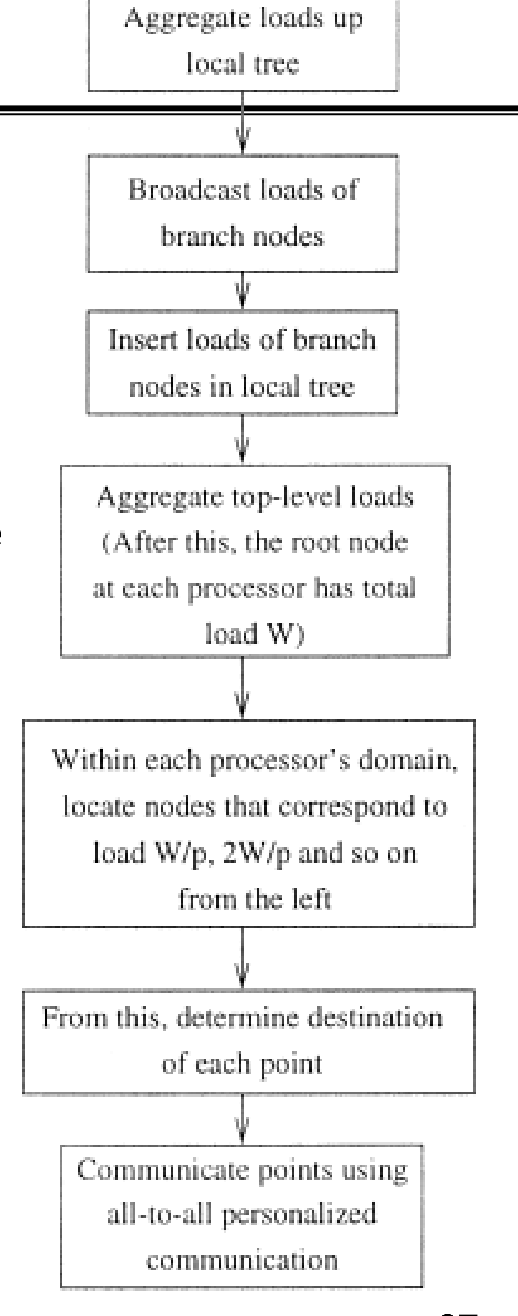
- Method 2: Static Partitioning, Dynamic Assignment (SPDA)
- Partitioning the domain into r subdomains or clusters as before
- Follow dynamic load balancing at each step based on the loads of the subdomains
- E.g.: Morton Ordering

Load Balancing

- Method 3: Dynamic Partitioning, Dynamic Assignment (DPDA)
- Allow clusters/subdomains of varying sizes
- Each node in the tree maintains the number of particles it interacted with
- After force computation, this summed along the tree; the value of load at each node now stores the number of interactions with all nodes rooted at the subtree; the root node contains the total number of interactions, W, in the system
- The load is partitioned into W/p; the corresponding load boundaries are 0, W/p, 2W/p,…,(p-1)W/p
- The load balancing problem now becomes locating one of these points in the tree

Load Balancing

- Each processor traverses its local tree in an in-order fashion and locates all load boundaries in its subdomain
- All particles lying in the tree between load boundaries iW/p and (i+1)W/p are collected in a bin for processor i and communicated to processor i



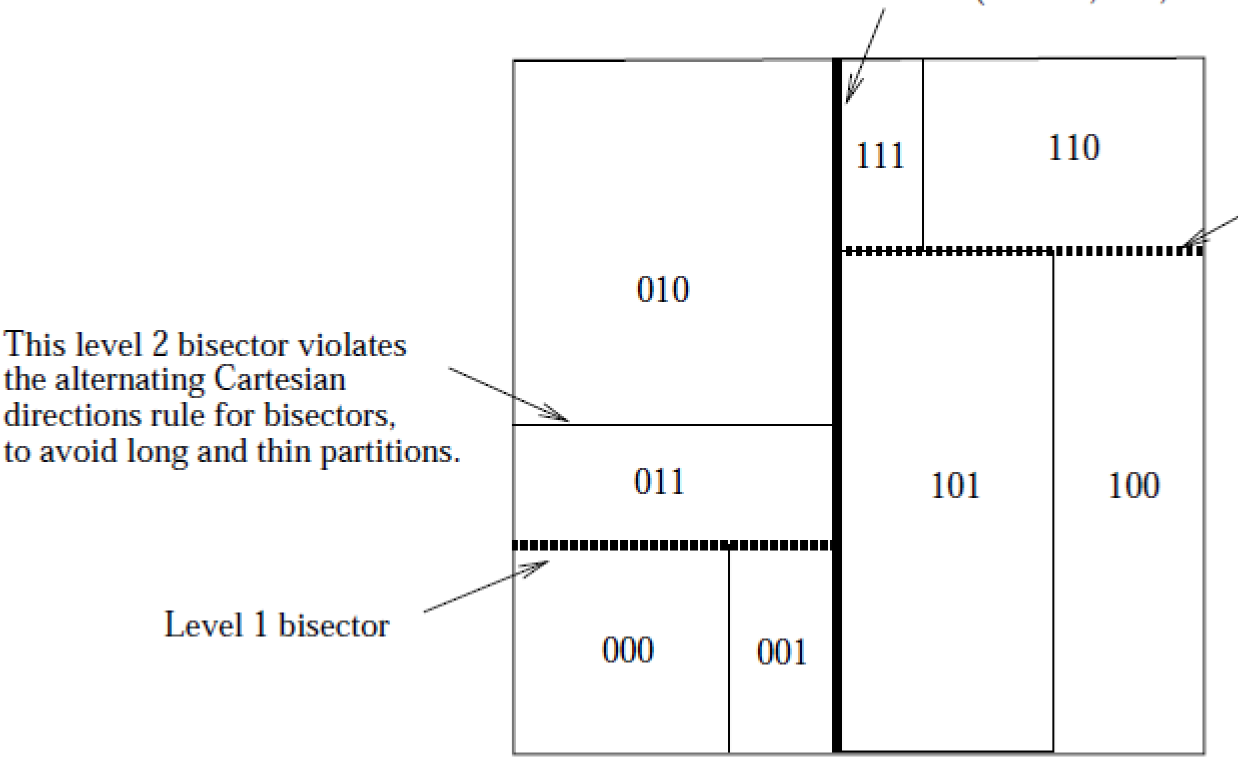
Load Balancing: Costzones

- DPDA scheme is also referred to as the costzones scheme
- The costs of computations are partitioned
- The costs are predicted based on the interactions in the previous time step
- In classical N-body problems, the distribution of particles changes very slowly across consecutive time steps
- Since a particle's cost depends on the distribution of particles, a particle's cost in one time-step is a good estimate of its cost in the next time step
- A good estimate of the particle's cost is simply the number of interactions required to compute the net force on that particle

Load Balancing: Costzones

- Partition the tree rather than partition the space
- In the costzones scheme, the tree is laid in a 2D plane
- The cost of every particle is stored with the particle
- Internal cell holds the sum of the costs of all particles that are contained within it
- The total cost in the domain is divided among processors so that every processor has a contiguous, equal range or zone of costs (hence the name costzones)
- E.g.: a total cost of 1000 would be split among 10 processors so that the zone comprising costs 1-100 is assigned to the first processor, zone 101-200 to the second, and so on.

Load Balancing: ORB (Orthogonal Recursive Bisection)



References

- The paper "Scalable parallel formulations of the Barnes-Hut method for n-body simulations" by Grama, Kumar and Sameh. In Supercomputing 1994.
- The paper "Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, Fast Multipole, and Dardiosity" by Singh, Holt, Totsuka, Gupta and Hennessey. In Journal of Parallel and Distributed Computing, 1994