Parallel N-Body simulations

Outline

- Motivation
  - Importance of N-body problems. N(N-1)/2 pairs, In general $O(N^2)$ work
- Goal for high performance:
  - Reduce the number of particles in the force sum
  - Trade off: reduce accuracy and increase speed?
- Straightforward approach
  - Example: Particles-in-Cells
- Basic Data Structures: Quad Trees and Oct Trees
- The Barnes-Hut Algorithm (BH)
  - An O(N log N) approximation algorithm for the N-Body problem
- The Fast Multipole Method (FMM)
  - An O(N) approximate algorithm for the N-Body problem

  For more information See http://www.cs.berkeley.edu/~demmel/
- Parallelizing BH, FMM and Particles-in-Cells

What is N-Body simulation?

An N-body simulation numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body.

Examples of applications:

- Astrophysics – evolution of galaxy
- Vortex particle simulation of turbulence
- Molecular Dynamics
- Plasma Simulation
- Electron-Beam Lithography Device Simulation

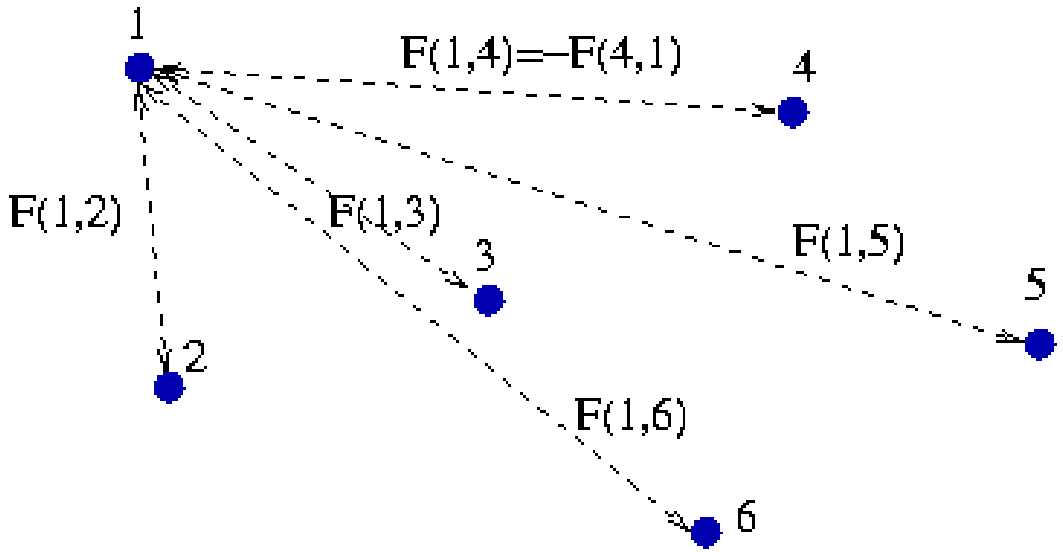
Applications

° **Astrophysics and Celestial Mechanics - 1992**

- Intel Delta = 1992 supercomputer, 512 Intel i860s
- 17 million particles, 600 time steps, 24 hours elapsed time
  - M. Warren and J. Salmon
  - Gordon Bell Prize at Supercomputing 1992
- 1% accuracy
- Direct method (17 Flops/particle/time step) at 5.2 Gflops would have taken 18 years, 6570 times longer

° **Vortex particle simulation of turbulence – 2009**

- Cluster of 256 NVIDIA GeForce 8800 GPUs
- 16.8 million particles
  - T. Hamada, R. Yokota, K. Nitadori. T. Narumi, K. Yasoki et al
  - Gordon Bell Prize for Price/Performance at Supercomputing 2009



For time step from time t to t+dt do

For each particle i do

Compute the total force acting on i:

$F(i) = F(i,1) + \ldots + F(i,i-1) + F(i,i+1) \ldots + F(i,n)$

Compute acceleration of particle i:

$a(i) = F(i)/Mass(i);$

Compute new position of i at time t+dt:

$P(i) := P(i) + 0.5*a(i)*dt*dt$

Particle Simulation, general

Basic structure :

- Essential is the time-stepping
- For " each time" evaluate force on all particles

```
t = 0
while t < t_final
    for i = 1 to n                                  … n = number of particles
       compute f(i) = force on particle i           … nested inner loop

    for i = 1 to n
       move particle i by force f(i) for time dt    … using F = m a

    compute properties of particles (like energy, angular momentum)
    t = t + dt
end while
```

Particle Simulation, bottlenecks

° **Computational bottlenecks?**
- **force calculation; contains a nested loop** — **Yes**
- **move operation; fixed amount of work per particle** — **No**
- **calculation of properties** — **Possible**

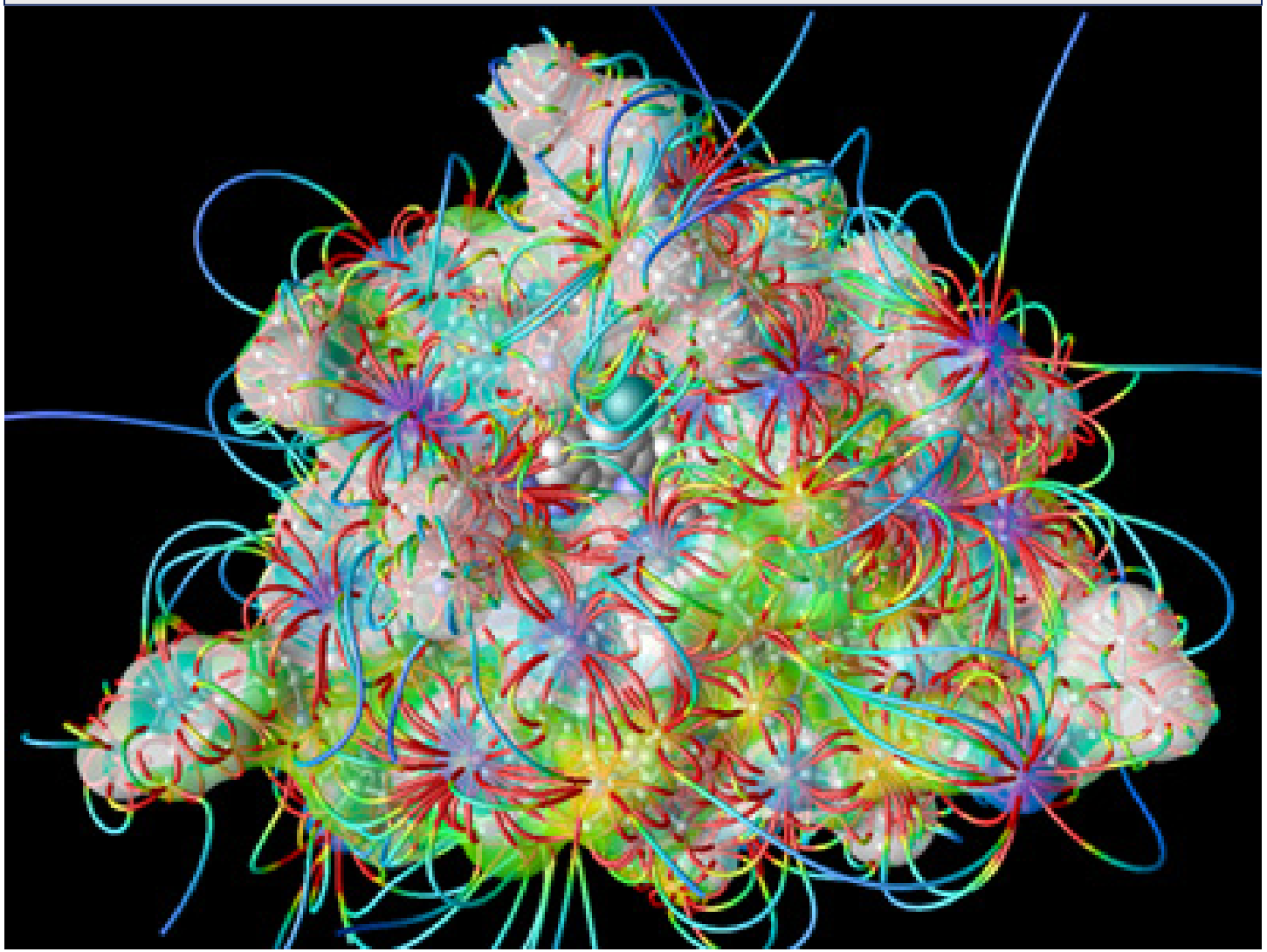
° **For high performance, focus on bottleneck force calculation**

° **Types of force to distinguish (in practice)** — **(in parallel code)**
- **External force** — **Trivial**
  - **Is independent on the other particles**
- **Short-range force** — **Easy**
  - **Depends on particles within fixed range**
- **Long-range force** — **Difficult**
  - **Depends on all other particles**

Particle Simulation, bottleneck

- **f(i) = external_force + short-range_force + long-range_force**
  - **External_force is embarrassingly parallel and costs O(N) for all particles**
    - **Satellites in gravitational field of the earth.**
  - **Short-range_force requires interacting with one or a few neighbors, so still O(N)**
    - **van der Waals force, Yukawa**
    - **bouncing balls, problem: collision-times, quite different approach**
  - **Long-range_force (gravity or electrostatics) requires all-to-all interactions**
    - **$f(i) = \sum_{k \neq i} f(i,k)$ … f(i,k) = force on i from k**
    - **$f(i,k) = c * v / ||v||^3$ in 3 dimensions or**
    - **$f(i,k) = c * v / ||v||^2$ in 2 dimensions**
      - **v = vector from particle i to particle k , c = product of masses or charges**
      - **||v|| = length of v**
    - **Obvious algorithms costs $O(N^2)$, but we can do better…**
  - **Focus in class is on long-range force**
  - **In computerlab parallelization exercise for short-range force**

Example of challenging applications: Protein folding (IBM Blue Gene project: 1,000,000 processors)



Estimated computational effort (protein folding): 3 years to simulate 100 μs on a petaflop/s computer

Physical time for simulation	10^{-4} seconds
Typical time-step size	10^{-15} s
Number of MD time steps	10^{11}
#atoms in a typical protein and water simulation	32000
Approximate number of interactions in force calc.	10^{9}
Machine instructions per force calculation	1000
Total number of machine instructions	10^{23}

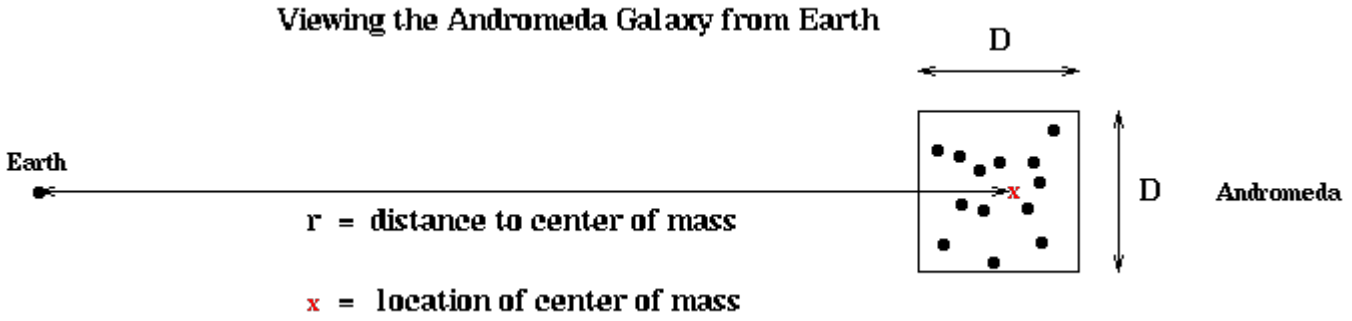
Note: Recent development AlphaFold: Using AI for scientific discovery

How to reduce # of particles in force sum?

- **Simple approach**
  - Ignore particles that are enough far away
- **Subscribe particles to cells, regions of space**
- **Particles in one region interact only with particles in nearby regions**
  - This makes a O(N) method. Particles in far away cells need not even be considered
  - Inaccurate or even incorrect for long-range forces like gravity.
  - Useful for short-range forces like Lennard-Jones in molecular dynamics.
    - Cell size related to range of interaction

Reducing the number of particles in the force sum

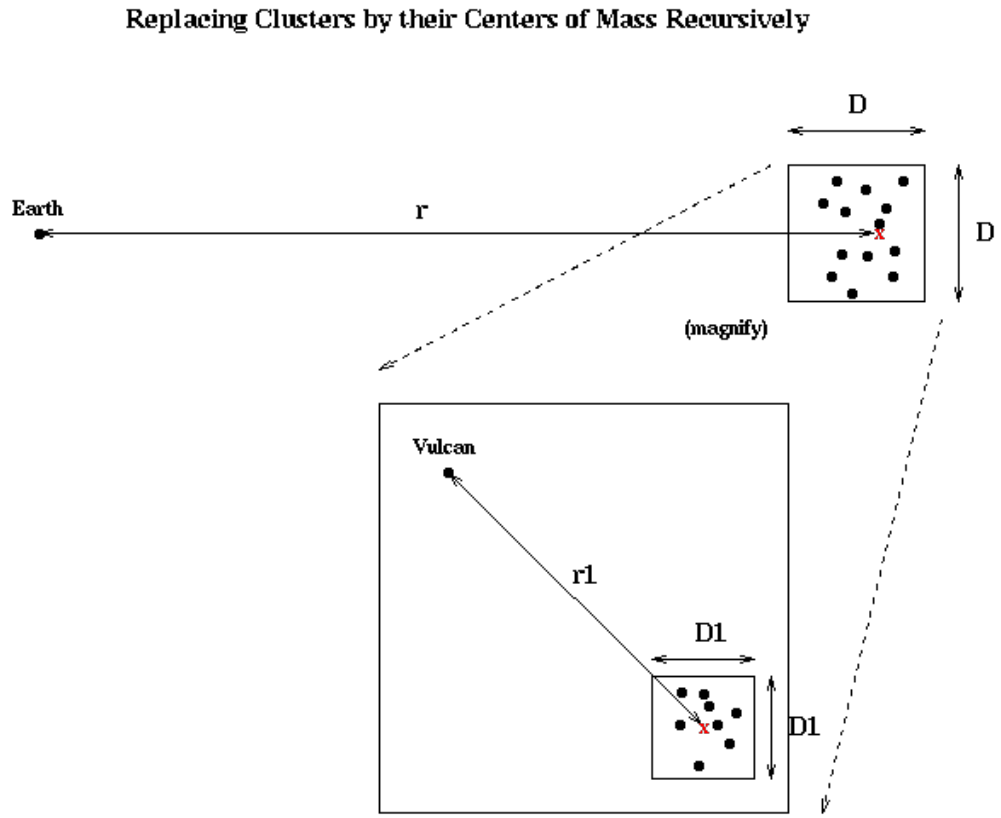
- All later divide and conquer algorithms use same intuition
- Consider computing force on earth due to all celestial bodies
  - Look at night sky, # terms in force sum ≥ number of visible stars
  - One "star" is really the Andromeda galaxy, which contains billions of real stars
    - Seems like a lot more work than we thought …
- OK to approximate all stars in Andromeda by a single point at its Center of Mass (CM) with the same total mass
  - D = size of box containing Andromeda , r = distance of CM to Earth
  - Require that D/r be "small enough"



- Old idea:Newton approximated earth and falling apple by Centres of Mass

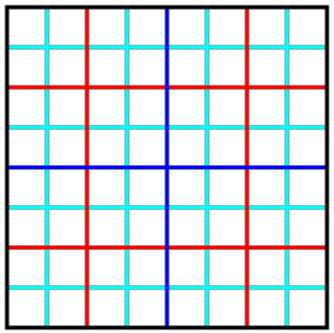
Using points at CM recursively

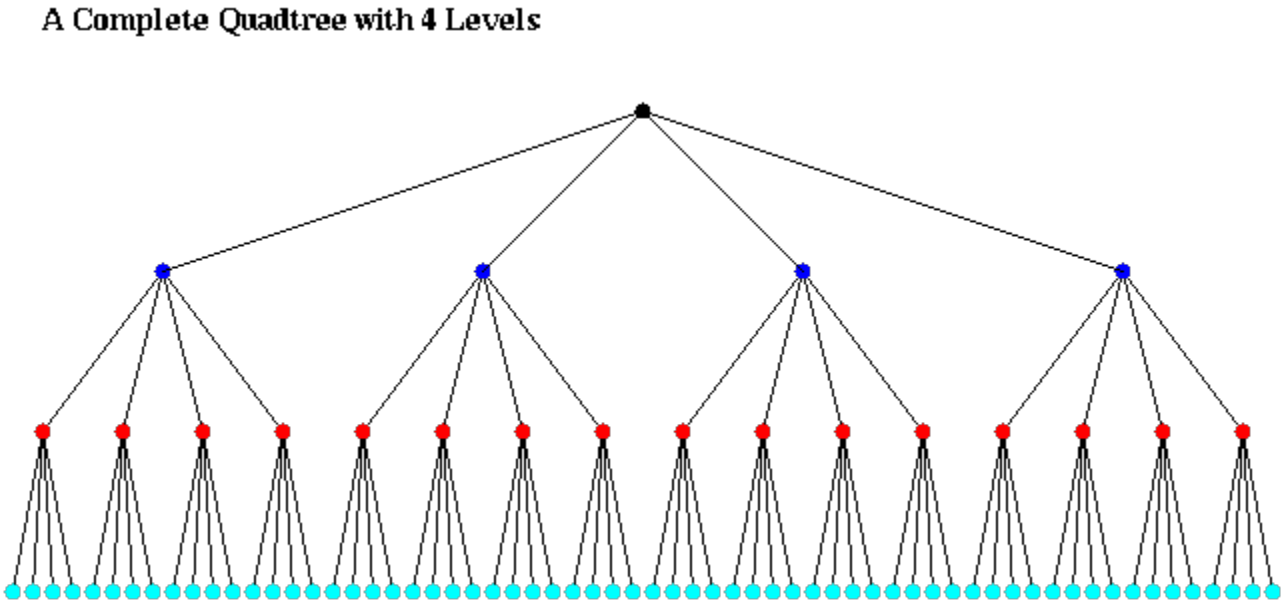
- From Andromeda's point of view, Milky Way is also a point mass
- Within Andromeda, picture repeats itself
  - As long as D1/r1 is small enough, stars inside smaller box can be replaced by their CM to compute the force on Vulcan
  - Boxes nest in boxes recursively



Quad Tree

- 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 non-leaf node has 4 children

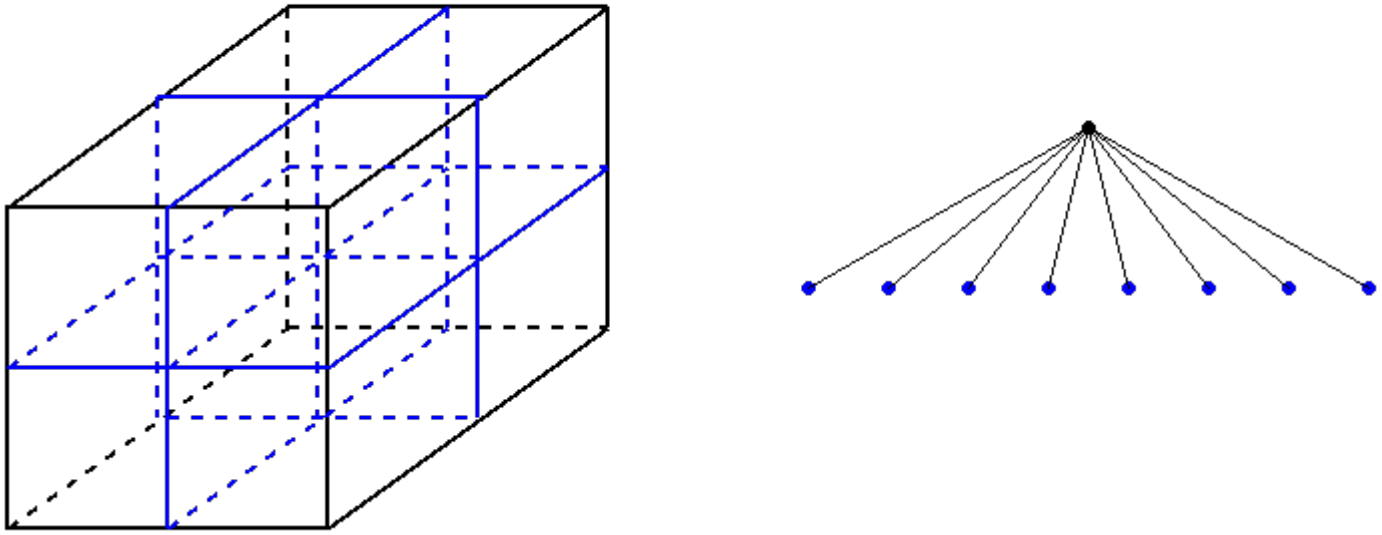




Oct Tree

° **Similar Data Structure to subdivide 3-d space**

2 Levels of an Octree



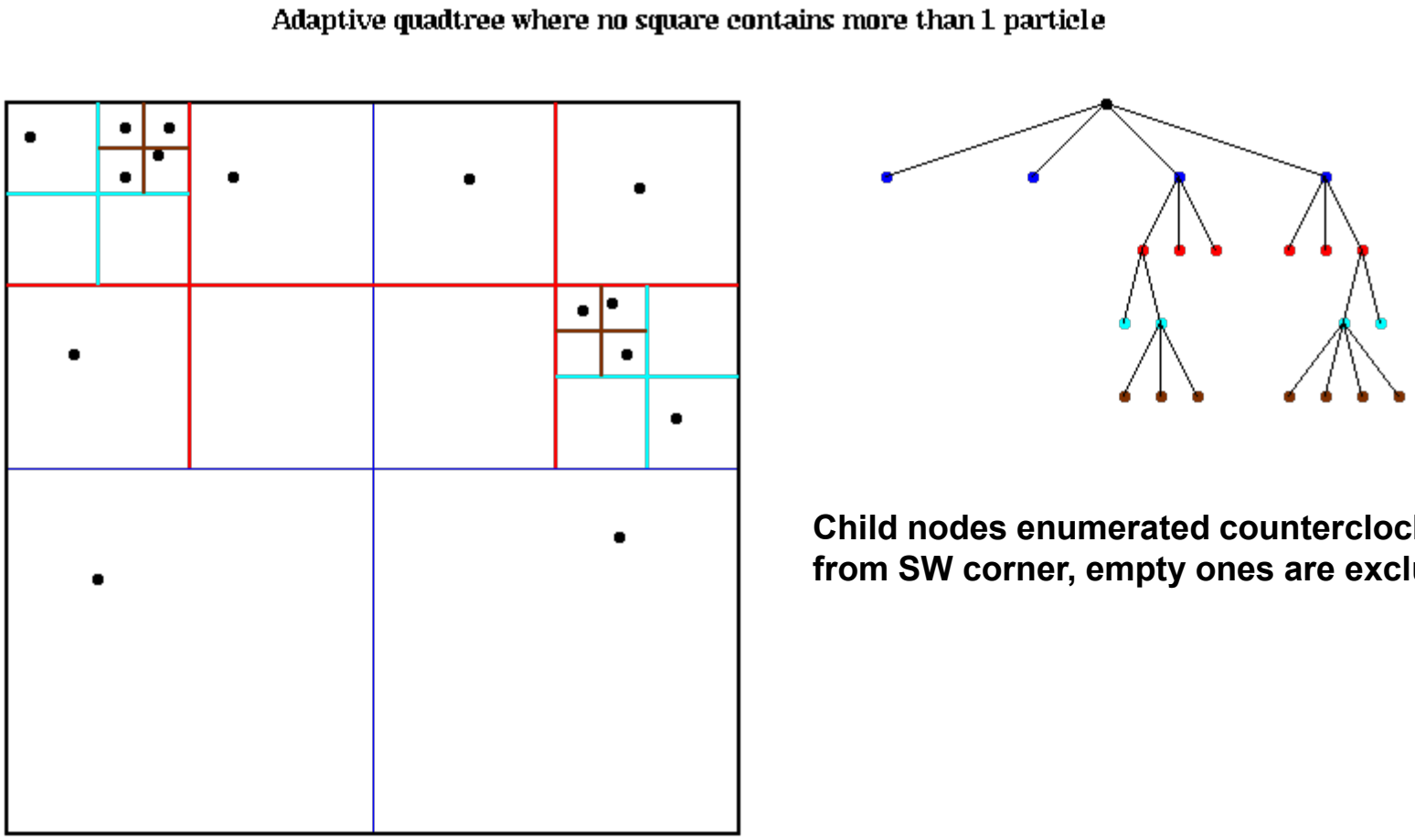
Using Quad Trees and Oct Trees

- All these algorithms begin by constructing a tree to hold all the particles

- Interesting cases have non-uniformly distributed particles
  - In a complete tree most nodes would be empty, a waste of space and time

- An Adaptive Quad- or Oct-tree only subdivides space where particles are located

Example of an Adaptive Quad Tree



Child nodes enumerated counterclockwise from SW corner, empty ones are excluded

Adaptive Tree Building Algorithm

- Tree building
  - Insert particles one by one at the root
  - Let the tree grow: Move particles from the root to an empty leave in the top of the tree, creating new leaves (nodes) if necessary
  - Remove empty leaves. All child-leaves of a node are created if only one of them is needed

- More details follow

Adaptive Quad Tree Algorithm (Oct Tree analogous)

```
Procedure QuadTreeBuild
  QuadTree = {emtpy}
   for j = 1 to N                          … loop over all N particles
       Quad_Tree_Insert(j, root)           … insert particle j in QuadTree
   endfor
   …  At this point, each leaf of Quad_Tree will have 0 or 1 particles
   …  There will be 0 particles when some sibling has 1
   Traverse the QuadTree eliminating empty leaves  … via, say Breadth First Search

Procedure Quad_Tree_Insert(j, n) … Try to insert particle j at node n in Quad_Tree
   if n is an internal node                … n has 4 children
      determine which child c of node n contains particle j
      Quad_Tree_Insert(j, c)
  else if n contains 1 particle  …  n is a leaf      Easy change for  q > 1 particles/leaf
      add n's 4 children to the Quad_Tree
      move the particle already in n into the child containing it
      let c be the child of n containing j
      Quad_Tree_Insert(j, c)
   else                                    …  n empty
      store particle j in node n
   end
```

Adaptive Quad Tree Building Cost

°Cost ≤ N * maximum cost of a Quad_Tree_Insert
= 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 coordinates),
so Cost = O(b N)

- Note: depth of QuadTree b ≥ 1 + ^{4}log N

Barnes-Hut Algorithm

- "A Hierarchical O(n log n) force calculation algorithm",
  - J. Barnes and P. Hut, Nature, v. 324 (1986), and many later papers

- Good for low accuracy calculations: Typically

$$\text{RMS error} = (\Sigma_k \,||\, \text{approx } f(k) - \text{true } f(k) \,||^2 \,/\, ||\, \text{true } f(k) \,||^2 / N)^{1/2}$$

$$\sim 1\%$$

(other measures better if some true force f(k) ~ 0)

- Accuracy can easily improved, by changing a single parameter θ, but the cost increases as well

Barnes-Hut Algorithm

° **High Level description:**

1) Build the QuadTree using QuadTreeBuild
… already described, cost = O(N log N) or O(b N)

2) For each node, a sub-square 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) or O(b N)

3) For each particle, traverse the QuadTree to compute the force on it, using the CM and TM of "distant" sub-squares
… core of algorithm
… cost depends on accuracy desired but still O(N log N) or O(bN)

Total cost: still O(N log N) or O(bN)

→ Achieved order reduction per time step: $O(N^2)$ to O(N logN)

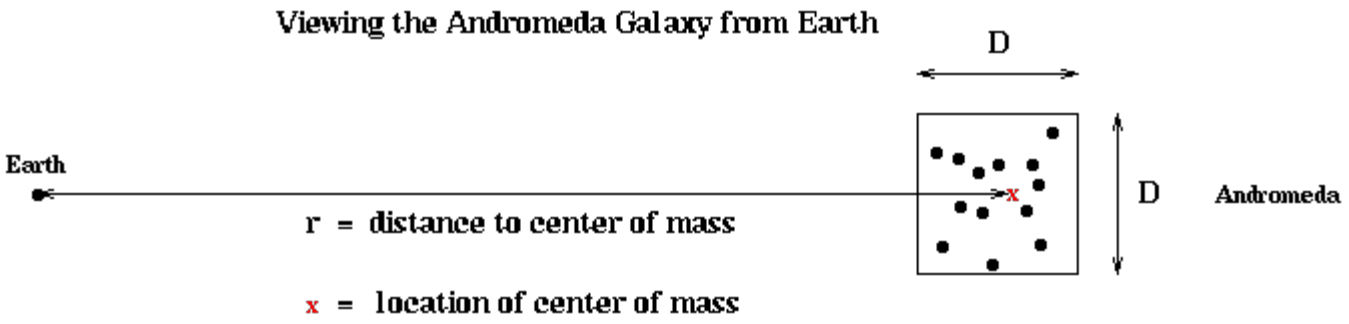
Step 2 of BH: Compute CM and total mass of each node

```
… Compute 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 all children before their parent
        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
```

Step 3 of BH: compute force on each particle

- **Idea: For each node (=square), 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 away enough" from the particle**
- **For each particle, use as few nodes as possible to compute force, subject to an accuracy constraint**
- **Need criterion to decide if node is far enough from particle**
  - **D = side length of node**
  - **r = distance from particle to CM of node**
  - **θ = user supplied error tolerance (usually < 1)**
  - **Use CM and TM to approximate force of node (and thus all particles within it) on particle if D/r < θ**



Force on a particle due to a node

° Suppose a node n, with CM and TM, and a particle k,

- 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 (node n)
- Let TM be the total mass of node n
- $r = ((x_k - x_{CM})^2 + (y_k - y_{CM})^2 + (z_k - z_{CM})^2)^{1/2}$
- G is gravitational constant

° Assume that $D / r < \theta$:

Force on k ~ $G * m * TM * \{x_{CM} - x_k , y_{CM} - y_k , z_{CM} - z_k\} / r^3$

° If $D / r \geq \theta$ calculate force on k for all children of n

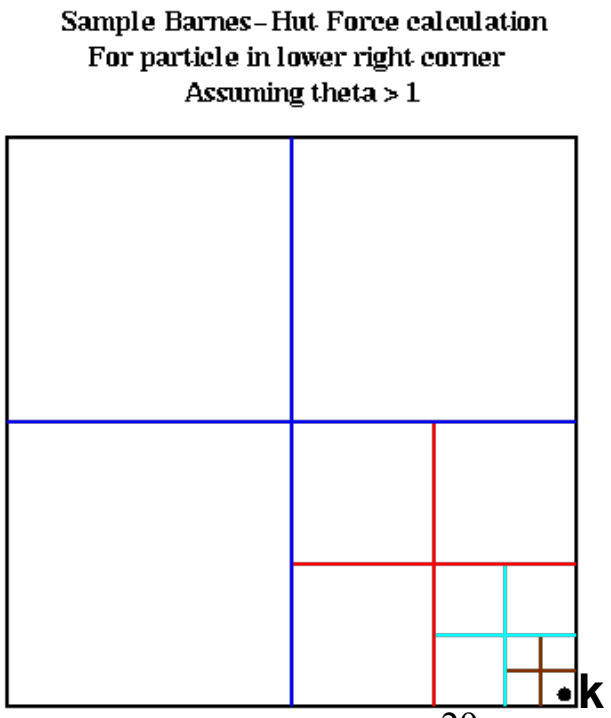
Step 3 of BH: force calculation

```
… 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 (except k)
endfor

function f = TreeForce( k, n )
   … compute force on particle k due to all particles inside node n (except k)
   f = 0
   if n contains one particle (not k)  … evaluate directly
      f = force computed using standard formula between two particles
   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
```

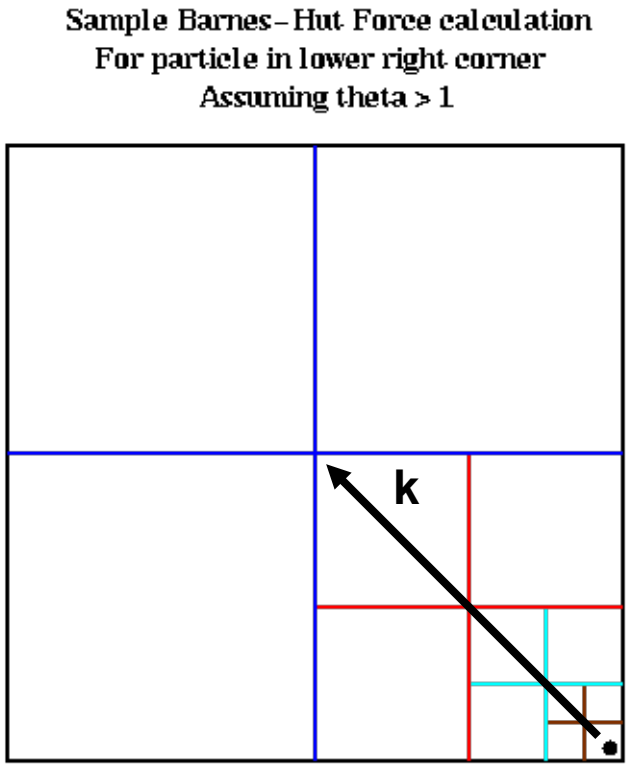
Step 3 of BH: Analysis

- **Correctness: 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"; Example: Assume $\theta = 1$
    - For each undivided node, see fig., (except one containing k), $D/r < 1 < \theta$
    - There are only 3 nodes to consider at each level of the QuadTree, see fig.
    - 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



Barnes-Hut: Summary

- Total cost = O(Σ_k level of k) = O(N log N). Strongly depends on θ
- If θ becomes 0, total cost is O(N^2), since all nodes have to be considered for any k
- If θ <1, in each level a limited number of nodes, independent of N, needs to be considered.



- Challenge: Parallelization of BH

Alternative approaches

- **Barnes-Hut: Composite nodes contain only "first order" information about constituents**
  - **Total mass and the Centre-of-mass**
  - **Higher order information may be maintained as well**
- **Greengard and Rokhlin used multipole information, of the mass- or charge distribution in a clever way: Fast Multipole method**
  - **Use of trees and the divide-and-conquer idea the same way as in Barnes-Hut**

Fast Multiple Method (FMM)

- "A fast algorithm for particle simulation", L. Greengard and V. Rokhlin, J. Comp. Phys. V. 73, 1987, many later papers

- Differences from Barnes-Hut
  - FMM computes the *potential* at every point, not just the force
  - FMM uses more information in each box than the CM and TM, so it is both more accurate and more expensive
  - In compensation, FMM accesses a fixed set of boxes at every level, independent of D/r
  - BH uses fixed information (CM and TM) in every box, but # boxes increases with accuracy.
  - FMM uses a fixed # boxes, but the amount of information per box increase with accuracy.

Parallelizing Hierarchical N-Body codes

- Barnes-Hut, FMM and related algorithms have similar computational structure:
  1) Build the QuadTree
  2) Traverse QuadTree from leaves to root and build outer expansions (just (TM,CM) for Barnes-Hut)
  3) Traverse QuadTree from root to leaves and build any inner expansions
  4) Traverse QuadTree to accumulate forces for each particle

- QuadTree changes dynamically when the particles move, so the tree has to be rebuilt (or adjusted) every time step.
- But: No doubly nested loop over all particles anywhere in the algorithm
- All 4 phases have to be parallelized efficiently.

Parallelizing Hierachical N-Body codes

- **General idea: Domain decomposition**
  - Assign regions of space to each processor
  - Regions may have different size or shape, to get a good load balance
    - Each region will have about N/p particles
  - Each processor will store part of QuadTree containing all particles (=leaves) in its region, and their ancestors in QuadTree
    - Root of tree and some generations stored by all processors, nodes may also be shared
  - Each processor will also store adjoining parts of QuadTree needed to compute forces for particles it owns
    - Subset of QuadTree needed by a processor called the Locally Essential Tree (LET)
  - Given the LET, all force accumulations (step 4)) are done in parallel, without communication
- **Coarse grained parallelism**
  - Each domain solves its own N-body problem, but somehow has to take into account the effect of all the other particles as well; like the ghost-points in the Poisson problem. Here those particles generate some "background" potential (in FMM)

Parallelizing Tree building

- **High performance is lost when each domain builds (or gets) the complete tree. It scales no longer with #processors, since the tree building has the same time complexity as other 3 phases in the computation.**

- **Domain decomposition: Assign regions of space to each processor. But how?**

Load balancing

° **Generic problem:**

Assign P groups of particles to P processes, such that these processes can evaluate the forces on its particles as fast as possible

° **Complications:**

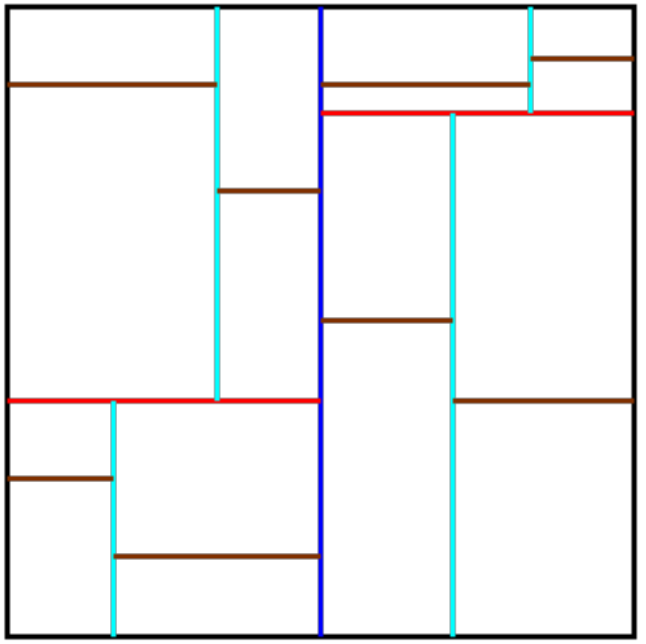
- **Particles move, and groups may change dynamically**
- **Cost of regrouping may be considerable**
- **Optimal strategy depends on architecture as well. Performance analysis is required**

° **Exploit the idea of locality. But how?**

Load Balancing Scheme 1

- Orthogonal Recursive Bisection (ORB) of space
  - Warren and Salmon, Supercomputing 92
- Recursively split region along axes into regions containing equal numbers of particles
  - Particles are grouped in rectangular regions; may be very elongated
  - No relation with tree

Partitioning for 16 procs:



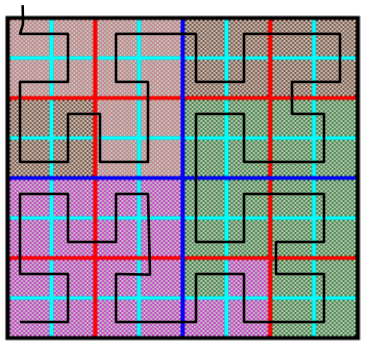
Load Balancing Scheme 2

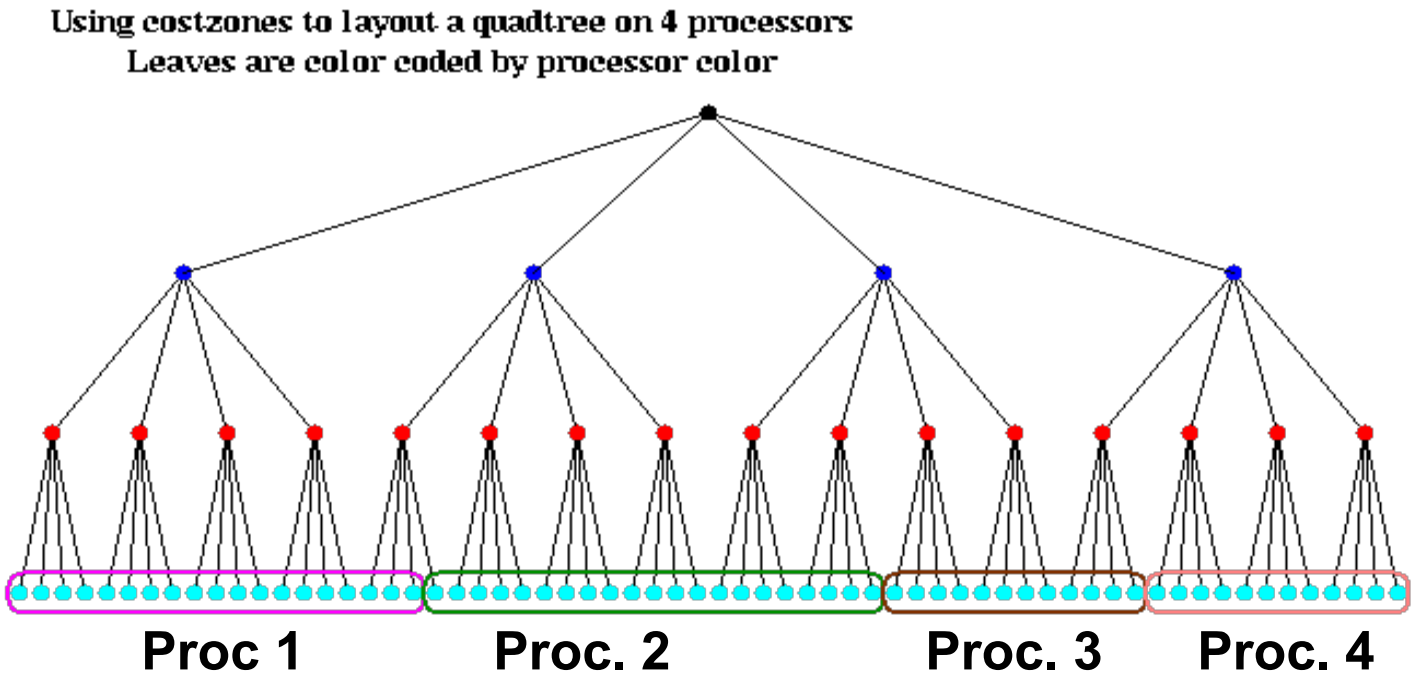
° Idea: Partition QuadTree instead of space
  - Estimate work for each node, call total work W
  - Arrange nodes of QuadTree in some linear order (lots of choices)
  - Assign contiguous groups of nodes with work W/p to processors

° Method called: Costzones or Hashed Tree
  - J.P. Singh, PhD thesis, Stanford, 1993
  - Warren and Salmon, Supercomputing 93

Load Balancing Scheme 2

- **Make sure that neighboring leaves in the tree are also neighboring in space**
  - **Which of the 4 children of a node is "first" depends on the position of the node**
  - **Orientation changes: clockwise / counter-clockwise**





Implementing Costzones

- **Problem: Partitioning of QuadTree implies that the tree already exists**
- **But the tree building phase is a computational bottleneck**
- **Not practical to first compute QuadTree, in order to compute Costzones, to then determine how to best build QuadTree in parallel**

Implementing Costzones

- **Random Sampling**
  - All processors own some particles,
  - All processors send a small random sample of their particles to Processor 1
  - Processor 1
    - builds small Quadtree serially,
    - determines its Costzones, and
    - broadcasts them to all processors
  - Other processors build the part of Quadtree they are assigned by these Costzones

- **All processors know all Costzones**

- **This is needed later to compute LET's**

Locally Essential Trees (LETs)

- Warren and Salmon, 1992; Liu and Bhatt, 1994
- Definition:
  - A LET of a process is that part of the Tree that is necessary to compute the force on the particles that are owned by that process.
- Information about nodes near the root of the tree is present in all processes
- Information about nodes near some leaves of the tree that are all owned by a single process is needed in one or a few processes

Computing Locally Essential Trees (LETs)

- Warren and Salmon, 1992; Liu and Bhatt, 1994
- Every processor needs a subset of the whole QuadTree, called the LET, to compute the force on all particles it owns
- Shared Memory
  - Receiver driven protocol
  - Each processor reads part of QuadTree it needs from shared memory on demand, keeps it in cache
  - Drawback: cache memory appears to need to grow proportionally to P to remain scalable
- Distributed Memory
  - Sender driven protocol
  - Each processor decides which other processors need parts of its local subset of the Quadtree, and sends these subsets

Locally Essential Trees in Distributed Memory

° **Barnes-Hut**

Which nodes are needed?

- Let j and k be processes, n a node on process j
- Let D(n) be the side length of n
- Let $r_k(n)$ be the shortest distance from n to any point owned by k
- If either

  (1) $D(n)/r_k(n) < \theta$ and $D(\text{parent}(n))/r_k(\text{parent}(n)) \geq \theta$, or

  (2) $D(n)/r_k(n) \geq \theta$

  then node n is part of k's LET, and so process j should send n to k

- Condition (1) means (TM,CM) of n can be used on process k, but this is not true of any ancestor
- Condition (2) means that we need the ancestors of type (1) nodes too

Performance Results - 1

° 512 Proc Intel Delta

- Warren and Salmon, Supercomputing 92
- 8.8 10^6 particles, uniformly distributed
- .1% to 1% RMS error
  - Decomposing domain 7 s
  - Building the OctTree 7 s
  - Tree Traversal 33 s
  - Communication during traversal 6 s
  - Force evaluation 54 s
  - Load imbalance 7 s

Total 114 seconds = 5.8 Gflops

- Rises to 160 secs as particle distribution becomes non-uniform

Performance Results - 2

° Cray T3E

- Blackstone, 1999
- 10^{-4} RMS error
- General 80% efficient on up to 32 processors
- Example: 50K particles, both uniform and non-uniform
  - preliminary results; lots of tuning parameters to set

	Uniform		Non-uniform	
	1 proc	4 procs	1 proc	4 procs
Tree size	2745	2745	5729	5729
MaxDepth	4	4	10	10
Time(secs)	172.4	38.9	14.7	2.4
Speedup		4.4		6.1
Speedup vs $O(n^2)$		>50		>500

° Future work - portable, efficient code including all useful variants

Some recent Barnes-Hut parallel implementations

References	Hardware specifications	Programming platform	No. of particles simulated	Hardware-level performance evaluation
Zhang et al. [12]	IBM Power5 cluster with 118 nodes (16 cores and 64 GB memory for each node)	UPC	2 M	X
Zhang et al. [18]	x86 Linux Infiniband cluster. Each node has two hex-core Intel Xeon 5650 CPUs running at 2.67 GHz	UPC, C++	1–4 M	X
Dinan et al. [19]	887-node IBM 1350 Cluster	UPC, MPI	150,000	X
Hamada et al. [20]	GPU cluster using 128 NVIDIA GeForce 8800GTS GPUs	C++, OpenMP	562 M	X
Javed et al. [6]	StarBug cluster consisting of 8 dual Intel Xeon 2.8 GHz processors with 2 GB RAM	C, Java	12,000	X
Winkel et al. [14]	288 K cores of IBM Blue Gene/P system JUGENE	Fortran, C	2 billion	X

Source: Badri Munier, et al., On the parallelization and performance analysis of Barnes–Hut algorithm using Java parallel platforms, Spring-Nature Applied Sciences, 2020