N-Body Simulations – Background

° Suppose the answer at each point depends on data at all the other points
  • Electrostatic, gravitational force
  • Solution of elliptic PDEs
  • Graph partitioning

° Seems to require at least $O(n^2)$ work, communication

° If the dependence on “distant” data can be compressed
  • Because it gets smaller, smoother, simpler...

° Then by compressing data of groups of nearby points, can cut cost (work, communication) at distant points
  • Apply idea recursively: cost drops to $O(n \log n)$ or even $O(n)$

° Examples:
  • Barnes-Hut or Fast Multipole Method (FMM) for electrostatics/gravity/…
  • Multigrid for elliptic PDE
  • ...

1
Particle Simulation

\[ f(i) = \text{external\_force} + \text{nearest\_neighbor\_force} + \text{N-Body\_force} \]

- External_force is usually embarrassingly parallel and costs \( O(N) \) for all particles
- Nearest_neighbor_force requires interacting with a few neighbors, so still \( O(N) \)
  - van der Waals, bouncing balls
- N-Body_force (gravity or electrostatics) requires all-to-all interactions
  - \( f(i) = \sum f(i,k) \) ... \( f(i,k) = \text{force on } i \text{ from } k \)

  - \( f(i,k) = \frac{c*v}{||v||^3} \) in 3 dimensions or \( f(i,k) = \frac{c*v}{||v||^2} \) in 2 dimensions
    - \( v = \text{vector from particle } i \text{ to particle } k \), \( c = \text{product of masses or charges} \)
    - \( ||v|| = \text{length of } v \)
  - Obvious algorithm costs \( O(n^2) \), but we can do better...

\[
\begin{align*}
  t &= 0 \\
  \text{while } t < t_{\text{final}} & \quad \ldots \quad n = \text{number of particles} \\
  & \quad \quad \text{for } i = 1 \text{ to } n \quad \ldots \quad n = \text{number of particles} \\
  & \quad \quad \quad \text{compute } f(i) = \text{force on particle } i \\
  & \quad \quad \text{for } i = 1 \text{ to } n \\
  & \quad \quad \quad \quad \text{move particle } i \text{ under force } f(i) \text{ for time } dt \quad \ldots \quad \text{using } F=ma \\
  & \quad \quad \quad \quad \text{possibly compute interesting properties of particles (energy, etc.)} \\
  & \quad t = t + dt \\
\end{align*}
\]
**Force computation**

```plaintext
for i = 1 to n               ... n = number of particles
    for j = i+1 to n
        df = eval_force(i,j)
        f(i) = f(i) + df
        f(j) = f(j) - df
```

° **Parallellisation:**
  - Updates of forces are atomic operations
  - Parallellisation over outer loop: cyclic assignment to threads

How to implement the atomic updates?

  - Use locks? Probably not a good idea….
  - Use private buffers in one way or another? Listen to Licentiate Seminar by Marcus Holm later today
Applications (1/2)

° Astrophysics and Celestial Mechanics - 1992
  • Intel Delta = 1992 supercomputer, 512 Intel i860s
    - (Like "Jarmo´s", but one generation older, 16 cabinets)
  • 17 million particles, 600 time steps, 24 hours elapsed time
    – M. Warren and J. Salmon
    – Gordon Bell Prize at Supercomputing 1992
  • Sustained 5.2 Gigaflops = 44K Flops/particle/time step
  • “Barnes-Hut + costzones + LET”, 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
    – Gordon Bell Prize for Price/Performance at Supercomputing 2009
  • Sustained 20 Teraflops, or $8/Gigaflop
Applications (2/2)

° Molecular Dynamics
° Plasma Simulation
° Social Networks and Social Media
° …
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 $\geq$ number of visible stars
  - Oops! One “star” is really the Andromeda galaxy, which contains billions of real stars
    - Seems like a lot more work than we thought …
- Don’t worry, ok to approximate all stars in Andromeda by a single point at its center of mass (CM) with same total mass (TM)
  - $D =$ size of box containing Andromeda, $r =$ distance of CM to Earth
  - Require that $D/r$ be “small enough”

- Idea not new: Newton used it for earth and falling apple…
What is new: 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 $D_1/r_1$ 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

A Complete Quadtree with 4 Levels
Oct Trees

- Similar Data Structure to subdivide space
Using Quad Trees and Oct Trees

° All our algorithms begin by constructing a tree to hold all the particles

° Interesting cases have nonuniformly distributed particles
  • In a complete tree most nodes would be empty, a waste of space and time

° **Adaptive** Quad (Oct) Tree only subdivides space where particles are located
Example of an Adaptive Quad Tree

Adaptive quadtree where no square contains more than 1 particle

In practice, have $q>1$ particles/square; tuning parameter
Adaptive Quad Tree Algorithm (Oct Tree analogous)

Procedure Quad_Tree_Build
Quad_Tree = {empty}
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 Quad_Tree 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 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
Cost of Adaptive Quad Tree Construction

- Cost $\leq N \times \text{maximum cost of Quad\_Tree\_Insert} = O( N \times \text{maximum depth of Quad\_Tree})$

- **Uniform Distribution of particles**
  - Depth of Quad\_Tree = $O( \log N )$
  - Cost $\leq O( N \times \log N )$

- **Arbitrary distribution of particles**
  - Depth of Quad\_Tree = $O( \# \text{bits in particle coords } ) = O( b )$
  - Cost $\leq O( b N )$
Barnes-Hut Algorithm


° Good for low accuracy calculations:

\[
\text{RMS error} = \left( \sum_k \| \text{approx } f(k) - \text{true } f(k) \|^2 / \| \text{true } f(k) \|^2 / N \right)^{1/2}
\]

often \( \sim 1\% \)

(\text{other measures better if some true } f(k) \sim 0)

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

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

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 = \( O(N \log N) \) or \( O(bN) \)

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 \( O(N \log N) \) or \( O(bN) \)
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 of BH: compute force on each particle

° For each node = square, 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 away enough” from the particle

° For each particle, use as few nodes as possible to compute force, subject to accuracy constraint

° 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
  • $\theta =$ user supplied error tolerance $< 1$
  • Use CM and TM to approximate force of node on box if $D/r < \theta$

Viewing the Andromeda Galaxy from Earth

Earth

$D$

$r =$ distance to center of mass

$x =$ location of center of mass

Andromeda
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 direct formula
    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 direct formula
        else  ... need to look inside node
            for all children c of n
                f = f + TreeForce ( k, c )
            end for
        end if
    end if
end function
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 = $O(\text{level of } k)$
  - Total cost = $O(\sum_k \text{level of } k) = O(N \log N)$
    - Strongly depends on $\theta$
Parallelizing Hierarchical N-Body codes

Barnes-Hut, FMM and related algorithm 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

One parallelization scheme will work for them all

- Based on D. Blackston and T. Suel, Supercomputing 97
  - Assign regions of space to each processor
- Regions may have different shapes, to get 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
  - Top of tree stored by all processors, lower 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
Load Balancing Scheme 1: Orthogonal Recursive Bisection (ORB)

- Warren and Salmon, Supercomputing 92
- Recursively split region along axes into regions containing equal numbers of particles
- Works well for 2D, not 3D

Orthogonal Recursive Bisection

Partitioning for 16 procs:
Load Balancing Scheme 2: Costzones

° **Called Costzones for Shared Memory**
  - PhD thesis, J.P. Singh, Stanford, 1993

° **Called “Hashed Oct Tree” for Distributed Memory**
  - Warren and Salmon, Supercomputing 93

° **We will use the name Costzones for both**

° **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 blocks of nodes with work $W/p$ to processors
  - Works well in 3D

Using costzones to layout a quadtree on 4 processors
Leaves are color coded by processor color
Hashed QuadTrees (Warren and Salmon)

Assign unique key to each node in QuadTree, then compute hash(key) to get integers that can be linearly ordered.

If \((x,y)\) are coordinates of center of node, interleave bits to get key:
- Put 1 at left as "sentinel"
- Nodes near root of tree have shorter keys

**Building a key for a hashed Quadtree**

\[ x = 100101 \quad y = 110001 \]

**Assigning Keys to Quadtree Nodes**
Assign unique key to each node in QuadTree, then compute hash(key) to get a linear order

- key = interleaved bits of x,y coordinates of node, prefixed by 1

- Hash(key) = bottom h bits of key (eg h=4)

- Assign contiguous blocks of hash(key) to same processors
Determining Costzones in Parallel

- Not practical to compute QuadTree, in order to compute Costzones, to then determine how to best build QuadTree

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

- All processors know all Costzones; we need this later to compute “Local Essential Trees”
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

How does each processor decide which other processors need parts of its local subset of the Quadtree?

Barnes-Hut:

- Let $j$ and $k$ be processors, $n$ a node on processor $j$; Does $k$ need $n$?
- Let $D(n)$ be the side length of $n$
- Let $r(n)$ be the shortest distance from $n$ to any point owned by $k$
- If either
  1. $D(n)/r(n) < \theta$ and $D(\text{parent}(n))/r(\text{parent}(n)) \geq \theta$, or
  2. $D(n)/r(n) \geq \theta$
    then node $n$ is part of $k$’s LET, and so proc $j$ should send $n$ to $k$
- Condition (1) means (TM,CM) of $n$ can be used on proc $k$, but this is not true of any ancestor
- Condition (2) means that we need the ancestors of type (1) nodes too
512 Proc Intel Delta

- Warren and Salmon, Supercomputing 92, Gordon Bell Prize
- 8.8 M particles, uniformly distributed
- .1% to 1% RMS error
- 114 seconds = 5.8 Gflops
  - Decomposing domain 7 secs
  - Building the OctTree 7 secs
  - Tree Traversal 33 secs
  - Communication during traversal 6 secs
  - Force evaluation 54 secs
  - Load imbalance 7 secs
- Rises to 160 secs as distribution becomes nonuniform
Performance Results - 2

\* Cray T3E

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

<table>
<thead>
<tr>
<th></th>
<th>Uniform</th>
<th>Nonuniform</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 proc</td>
<td>4 procs</td>
</tr>
<tr>
<td>Tree size</td>
<td>2745</td>
<td>2745</td>
</tr>
<tr>
<td>MaxDepth</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Time (secs)</td>
<td>172.4</td>
<td>38.9</td>
</tr>
<tr>
<td>Speedup</td>
<td>4.4</td>
<td></td>
</tr>
<tr>
<td>Speedup vs $O(n^2)$</td>
<td>&gt;50</td>
<td></td>
</tr>
</tbody>
</table>

\* Ultimate goal - portable, tunable code including all useful variants