The N-body Problems

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

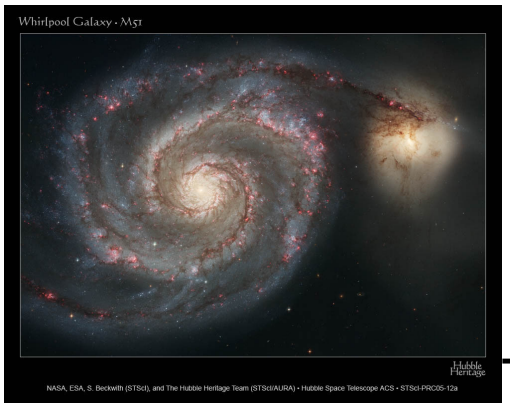
High Performance Computing I, 2012

Motivation

The so-called **N-body** problem has become almost synonymous with gravitation. It is, however, much more generically applicable than that:

- Gravitation
- Electrostatics/Magnetostatics
- Elliptic Partial Differential Equations (PDEs), more generally
- Graph Partitioning

Principle Concepts



The main concept involved in the **N-body** problem is pretty simple:

- When you are talking about N particles, you have **long-ranged** forces that require you to perform summations of $\mathcal{O}(N^2)$ work, and "considerable" communication.
- More generally than that, **locality** is not inherently present - e.g. in electrostatics you have a position dependent potential

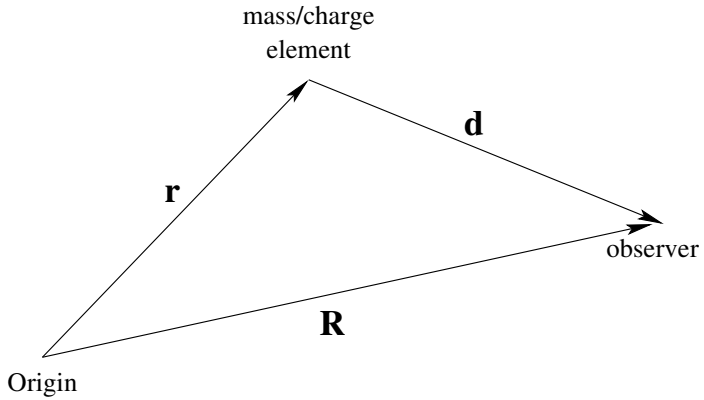
Simplifying

So how can we address this $\mathcal{O}(N^2)$ issue? The key to reducing the complexity is using the distant data in an approximate way:

- Think about gravity or electrostatics again - the farther away you are from a collection of (charged) bodies, the "simpler" they appear
- Or, in other words, the details get "smoothed over" by the relative distance
- This idea is basically the same as the concept of a multipole expansion (remember that?).

Multipole Revisited

Not to bore everyone to tears, but let's recapitulate the physical concept of a multipole expansion. Consider the usual picture:



Applying the law of cosines gives:

$$d^2 = R^2 + r^2 - 2rR\cos\theta$$

which we then approximate for $R >> r$,

$$1/d = \frac{1}{R}\left(1 - 2\epsilon x + \epsilon^2\right)^{-1/2},$$

and $\epsilon = r/R$, $x = \cos\theta$.

but, recall that the generating function for Legendre polynomials is

$$\left(1 - 2\epsilon x + \epsilon^2\right)^{-1/2} = \sum_{n=0}^{\infty} \epsilon^n \mathrm{P}_n(x),$$

so our distance expression becomes

$$1/d = \sum_{n=0}^{\infty} \frac{r^n}{R^{n+1}} \mathrm{P}_n(\cos\theta),$$

and a potential field is given by integration over (possibly continuous) sources,

$$V = \sum_{n=0}^{\infty} V_n = \sum_{n=0}^{\infty} \frac{1}{R^{n+1}} \int d^3\mathbf{r}\rho(\mathbf{r}) r^n \mathrm{P}_n(\cos\theta).$$

Distance Simplification

Using the same idea of "compressing" data in local neighborhoods, which becomes a good approximation as we view these regions from a large distance, we can gain considerable advantage (to $\mathcal{O}(N \log N)$ or even $\mathcal{O}(N)$). Some examples:

- Barnes-Hut (BH) Algorithm
- Fast Multipole Method (FMM)
- Multigrid Methods for Elliptic PDEs (another topic...)
- Multilevel graph partitioning (e.g. METIS)

We are going to cover the first two topics, and focus on particles/forces. First, though, we are going to cover some more elementary stuff ...

Force Models

There a slew of available force models, of course, so let us first carefully state our assumptions: we are going to deal with **pairwise** forces, and explicitly ignore so-called **many-body** effects. This assumption comprises the bulk of the simulation universe, but by no means all. Hopefully we will get a chance to come back to **many-body** considerations at some point in the future.

Classical Forces

In the classical Newtonian force model, then, we have a simple relation for pairwise forces:

$$\mathbf{F}(\mathbf{x}_i) = \sum_{j \neq i = 1}^{N} \mathbf{f}(\mathbf{x}_i, \mathbf{x}_j), \tag{1}$$

where the forces and positions are vector quantities, and this relation is for the force on particle i. The equation of motion should look pretty familiar:

$$\mathbf{F}(\mathbf{x}_i) = m_i \ddot{\mathbf{x}}_i. \tag{2}$$

Classical Verlet/Leap-Frog

Given our equation of motion in Eq. 2, stepping forward in time is simple enough: using superscripts to denote time steps,

$$\mathbf{x}_i^{k+1} = 2\mathbf{x}_i^k - \mathbf{x}_i^{k-1} + \delta t^2 \mathbf{F}(\mathbf{x}_i^k)/m_i, \tag{3}$$

Eq. 3 is due to Verlet and is otherwise known as the "Leap-Frog" algorithm[1]

[1] c.f. M. P. Allen and P. J. Tildesly, *Computer Simulation of Liquids*, (Clarendon Press, Oxford, 1994).

Scaling

Ok, so the time-stepping scheme is quite simple - a loop over time steps, combined with a force calculation that is $\mathcal{O}(N^2)$. Or in pseudo-code:

```
do it=1,num_time_steps
    do i=1,N
        f(1:3,i) = 0.0
        do j=1,N
            f(1:3,i) = f(1:3,i) + force_calc( pos(i), pos(j) )
        end do
    end do
end do
```

We can simplify a bit more than that ...

Simplifications

Even within this simple scheme, we can use what we know to considerably improve the scaling:

1. Newton's Third Law:

$$\mathbf{f}(\mathbf{x}_i, \mathbf{x}_j) = -\mathbf{f}(\mathbf{x}_j, \mathbf{x}_i)$$

2. Constraints - groups of particles moved together (bonds, etc.)
3. Cutoff Radii - forces become "negligible" past a certain distance

Newton's Third Law

So Newton's third law immediately cuts our work in half:

```
do it=1,num_time_steps
   do i=1,N
      f(1:3,i) = 0.0
      do j=i+1,N
         tmp(1:3) = force_calc( pos(i), pos(j) )
         f(1:3,i) = f(1:3,i) + tmp(1:3)
         f(1:3,j) = f(1:3,j) - tmp(1:3)
      end do
   end do
end do
```

Of course that comes at a price - it introduces a dependency between the two sets of loops (that hurts in terms of parallelization).

Force Replication

The concept of **force replication** is used to parallelize the force calculation that uses Newton's third law. Basically each process (or thread) gets a copy of the forces (hence the name), but is only responsible for keeping track of a **subset** of particles (i.e. the particles are *decomposed*). The partial sums then are pulled together to form the resulting true forces.

Pseudo-code for force replication:

```
do iproc=1,Nprocs
   do i=1,N
      partial_f(1:3,i,iproc) = 0.0
   end do
   do i = iproc, N, Nprocs  ! cyclic subdivision of particles
      dp j=i+1,N
         tmp(1:3) = force_calc( pos(i), pos(j) )
         partial_f(1:3,i,iproc) = partial_f(1:3,i,iproc) + tmp(1:3)
         partial_f(1:3,j,iproc) = partial_f(1:3,j,iproc) - tmp(1:3)
      end do
   end do
end do
do i=1,N
   f(1:3,i) = 0.0
   do iproc=1,Nproc
      f(1:3,i) = f(1:3,i) + partial_f(1:3,i,iproc)
   end do
end do
```

Note the cyclic decomposition of partial sums onto processors.

So what is hidden neatly inside that last piece of pseudo-code is the communication necessary to bring the partial sums together (unless you are using **OpenMP** or some other shared memory API). For very large N, though, the amount of computation should still dominate.

SHAKE Algorithm

Coming back to the idea of constraints (e.g. through molecular bonds), the very popular SHAKE algorithm deals with constraints by iteratively correcting the positions using *constraint forces*:

$$\Delta \mathbf{x}_i \simeq \frac{1}{m_i} \frac{\mu(d_{ij}^2 - \bar{d}_{ij}^2)}{2(\mathbf{d}_{ij} - \bar{\mathbf{d}}_{ij})} \mathbf{d}_{ij}, \tag{4}$$

where $\mathbf{d}_{ij}$ and $\bar{\mathbf{d}}_{ij}$ are the particle separations prior to the Verlet step, and after (unconstrained) respectively, with μ the reduced mass. These corrections are applied to all constrained particles repeatedly until a certain tolerance is reached.

J. P. Ryckaert, G. Ciccotti, and J. C Berendsen, J. Comp. Phys. **23**, 327-341 (1977).

Parallel SHAKE

Parallel SHAKE, as one might imagine, is made complex by the data dependencies. The usual approach is through spatial decomposition.

Cutoff Radii

A cutoff radius splits our original force calculation Eq. 1 into two pieces,

$$\mathbf{F}(\mathbf{x}_i) = \sum_{j:|\mathbf{x}_i - \mathbf{x}_j| < R_c} \mathbf{f}(\mathbf{x}_i, \mathbf{x}_j) + \sum_{j:|\mathbf{x}_i - \mathbf{x}_j| \geq R_c} \mathbf{f}(\mathbf{x}_i, \mathbf{x}_j) \tag{5}$$

where we are still taking into account the particles outside the cutoff radius, R_c, but potentially using a simpler (or less frequently updated) relation for outside particles. The scaling becomes

$$\mathcal{O}(nR_c^3 + \epsilon n^2), \tag{6}$$

and we will come back to **fast summation** methods for dealing with the second term.

Neighbor (Pair) Lists

Once a cutoff radius has been introduced, you also need neighbor lists to distinguish between particles within a distance R_c. A simple approach uses a linear array to store the neighbor list:

```
n0 = 0 ; nn = 0
do i=1,N
   do j=i+1,N
      dist_ij2 = (pos(1,i)-pos(1,j))**2+(pos(2,i)-pos(2,j))**2+(pos(3,i)-pos(3,j))**2
      if (dist_ij2 < Rcut*Rcut) then
         nn = nn + 1  ! nn is our basic counter over all neighbors
         neighbors(nn) = j
      end if
   end do
   N_neighbors(i) = nn - n0
   n0 = nn  ! n0 is the end poistion of the previous particle neighbors
end do
neigh_start(1) = 1
neigh_end(1) = N_neighbors(1)
do i=2,N
   neigh_start(i) = neigh_end(i-1)+1
   neigh_end(i) = neigh_end(i-1)+N_neighbors(i)
end do
```

Neighbor Lists in Parallel

Generating neighbor lists can be quite time-consuming; the preceding algorithm in parallel is made difficult by the dependence of the locations in the neighbor list itself. At the cost of using a bit of extra memory, we can maintain the neighbor list with a two-dimensional array:

```
do i=1,N
   ni = 0
   do j=i+1,N
      dist_ij2 = (pos(1,i)-pos(1,j))**2+(pos(2,i)-pos(2,j))**2+(pos(3,i)-pos(3,j))**2
      if (dist_ij2 < Rcut*Rcut) then
         ni = ni + 1
         neighbors2(ni,i) = j
      end if
   end do
   N_neighbors(i) = ni
end do
```

So the extra memory required in this two-dimensional scheme is just $N \cdot \mathrm{MAX}(\mathrm{N\_neighbors})$, while in the former one dimensional scheme we used $\mathrm{SUM}(\mathrm{N\_neighbors})$.

The hard part is coming up with `MAX(N_neighbors)` before carrying out the calculation (one can in practice use a maximum density argument).

and finally, working in a model where the neighbor lists are replicated across parallel processes:

```
!
!omp parallel do private(i,j,ni,dist_ij2)
do i=1,N,Nprocs
   ni = 0
   do j=i+1,N
      dist_ij2 = (pos(1,i)-pos(1,j))**2+(pos(2,i)-pos(2,j))**2+(pos(3,i)-pos(3,j))**2
      if (dist_ij2 < Rcut*Rcut) then
         ni = ni + 1
         neighbors2(ni,i) = j
      end if
   end do
   N_neighbors(i) = ni
end do
```

Note that, using OpenMP the load balancing is pretty trivial.

Force Calculations with Neighbor Lists

Using our neighbor lists, the force computation now looks like:

```
do it=1,num_time_steps
   do i=1,N
      f(1:3,i) = 0.0
      do j=1, N_neighbors(i)
         tmp(1:3) = force_calc( pos(i), pos(neighbors2(j,i)) )
         f(1:3,i) = f(1:3,i) + tmp(1:3)
         f(1:3,neighbors2(j,i)) = f(1:3,neighbors2(j,i)) - tmp(1:3)
      end do
   end do
end do
```

Load Balance and Communication Overhead

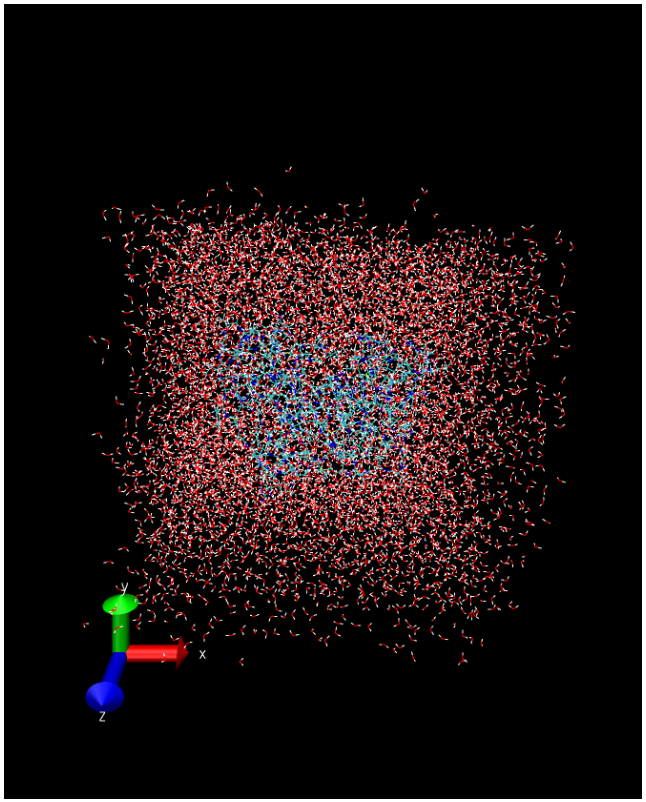
Using OpenMP load balance is quite easy to achieve - the replication scheme is a bit more involved due to the nature of the spatial/force decomposition.

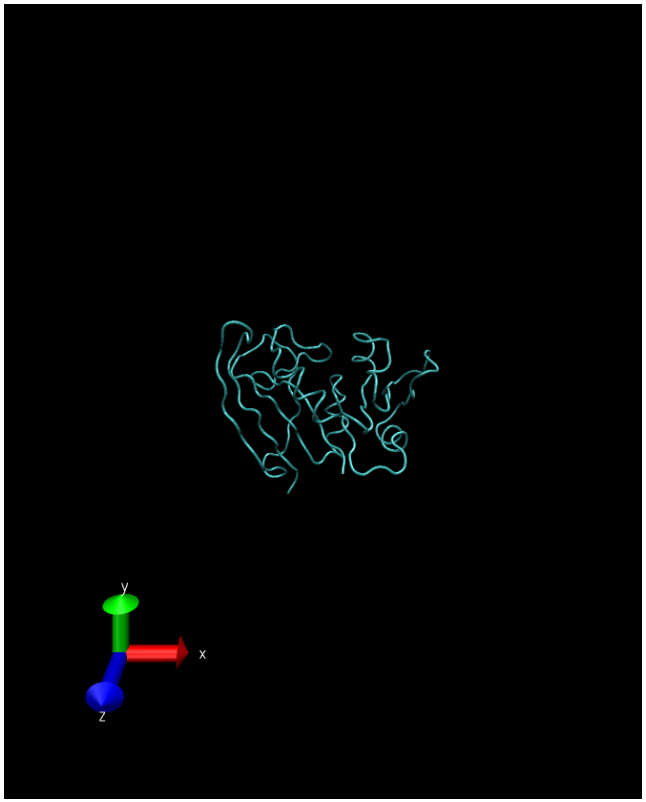
The average number of computations per parallel process is `N ave(N_neighbors)/P`, while the communication costs will be also be proportional to `N`. So the ever- important ratio of computation to communication is going to be

$$\frac{\text{ave(N\_neighbors)}}{\text{P}},$$

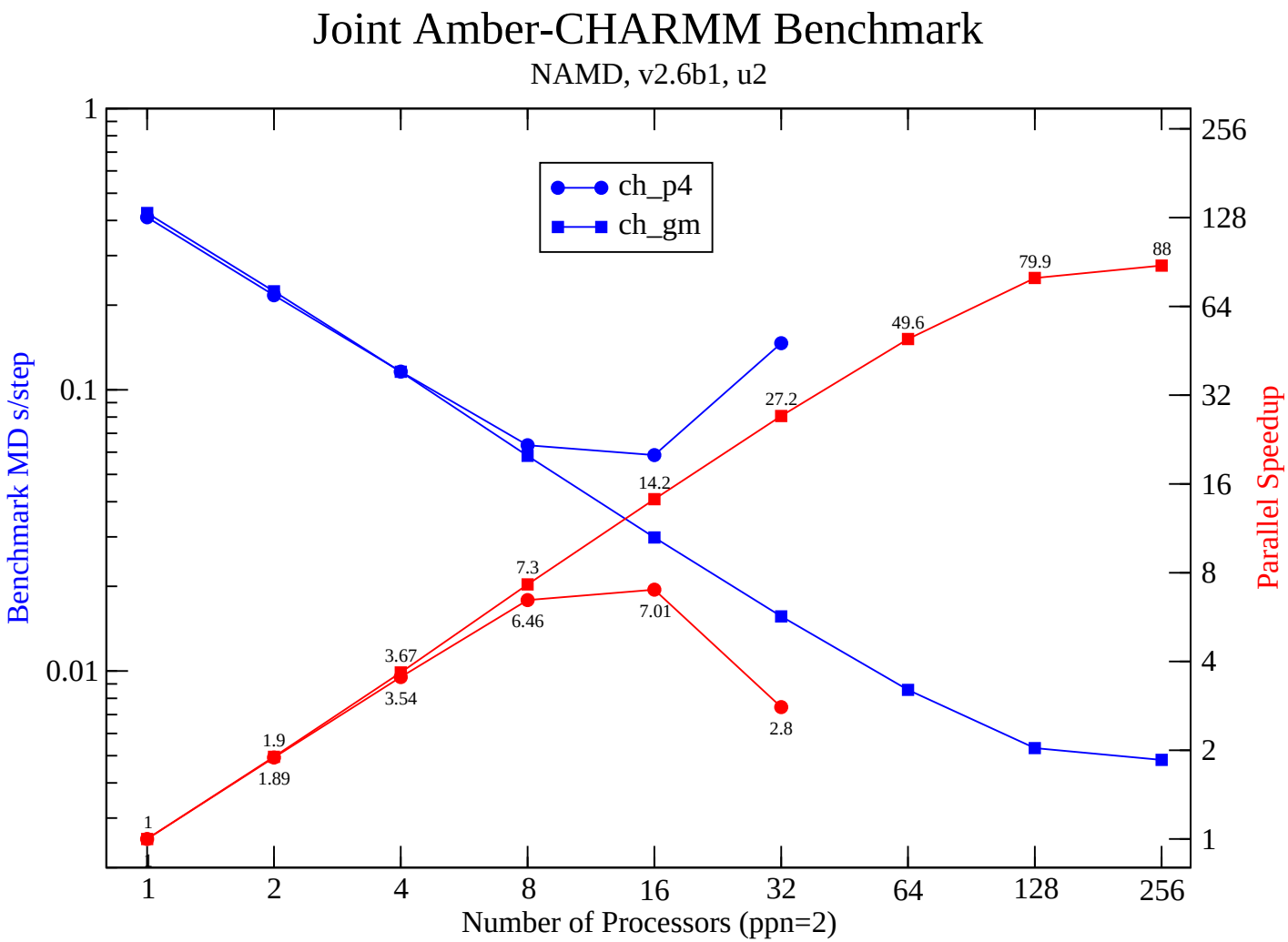
so not scalable. And in practice molecular dynamics codes are some of the most difficult codes in which to obtain good parallel performance.

JAC (Joint Amber-CHARMM) Benchmark





DHFR Protein, 7182 residues, 23558 atoms, 7023 TIP3 waters, PME (9Åcutoff)


Joint Amber-CHARMM Benchmark
NAMD, v2.6b1, u2
ch_p4
ch_gm
Benchmark MD s/step
Parallel Speedup
Number of Processors (ppn=2)
1
0.1
0.01
256
128
64
32
16
8
4
2
1
1
2
4
8
16
32
64
128
256
1
1.89
3.54
6.46
7.01
2.8
1.9
3.67
7.3
14.2
27.2
49.6
79.9
88

Ewald Sums

Another means to handle long-ranged (particularly electrostatic) forces:

- Long-range forces decrease slower than $f(r) \sim r^{-d}$ where d is dimensionality
- Ewald sums are alternative to cutoff radius
- Applied to PBC (periodic boundary conditions),

$$\mathbf{n} = (n_x L_x, n_y L_y, n_z L_z),$$

where the simulation cell has volume $L_x L_y L_z$.

Usual sum for Coulomb potential,

$$U = \sum_{\mathbf{n}} \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{q_i q_j}{|\mathbf{r}_{ij} + \mathbf{n}|}$$

which is only *conditionally convergent* (i.e. depends on summation order). The Ewald method breaks the above into two separate, rapidly converging sums.

- Apply screening (Gaussian) distribution to original point charge, to be summed in real space
- Apply equal and opposite counter-distribution, to be summed in reciprocal space

Ewald Decomposition

Using the trivial mathematical identity:

$$\frac{1}{r} = \frac{f(r)}{r} + \frac{1 - f(r)}{r},$$

and the main idea is to pick $f(r)$ "appropriately" to take into account the rapid variation at small r and the long range tail at large r.

- $f(r)/r$ should be negligible (better yet zero) beyond some cutoff value of r
- $(1 - f(r))/r$ should be smooth and slowly varying for all r, hence suitable for Fourier transform and approximation with a few Fourier terms

Choice of Function in Ewald

Wide amount of freedom in choosing $f(r)$ - tradition is the complementary error function[2]

$$\mathrm{erfc}(r) = \frac{2}{\sqrt{\pi}} \int_r^\infty dt \exp(-t^2),$$

and using this choice we obtain the following ....

[2] S. W. de Leeuw, J. W. Perram, and E. R. Smith, Proc. R. Soc. London Ser. A **373**, 27 (1980).

$$U = U^{(r)} + U^{(k)} + U^{(s)} + U^{(d)},$$

$$U^{(r)} = \frac{1}{2} \sum_{i,j}{}' \sum_{\mathbf{n} \in \mathcal{Z}^3} q_i q_j \frac{\mathrm{erfc}(\alpha |\mathbf{r}_{ij} + \mathbf{n}|)}{|\mathbf{r}_{ij} + \mathbf{n}|}$$

$$U^{(k)} = \frac{1}{2L^3} \sum_{k \neq 0} \frac{4\pi}{k^2} \exp\left(-k^2/4\alpha^2\right) |\tilde{\rho}(\mathbf{k})|^2$$

$$U^{(s)} = -\frac{\alpha}{\sqrt{\pi}} \sum_i q_i^2$$

$$U^{(d)} = \frac{2\pi}{(1+2\epsilon')L^3} \left(\sum_i q_i \mathbf{r}_i \right)^2$$

where

$$\tilde{\rho}(\mathbf{k}) = \int_{L^3} d^3 r \rho(r) e^{-i\mathbf{k}\cdot\mathbf{r}} = \sum_j q_j e^{-i\mathbf{k}\cdot\mathbf{r}_j}$$

is the usual Fourier transform of the charge density and the **dipole** term, $E^{(d)}$ assumes the PBC tends to an infinite cluster embedded in a medium of dielectric constant ϵ' (this helps deal with conditional covergence issues - it is independent of the **Ewald parameter**, α).

Ewald in Practice

Ewald sums are arguably the most common means of accurate treatment of long-ranged forces.

- Choose α large enough for *minimum image* convention (only interactions with nearest neighbors in short-range piece)
- Scales like $\mathcal{O}(N^{3/2})$
- Closely related alternatives, particle-mesh Ewald (PME), particle-particle-particle-mesh Ewald (P^3ME)

Particle Mesh Ewald (PME)

In the so-called particle mesh Ewald approach, the short range piece (screened) is treated directly, while the long-range piece is interpolated onto a mesh, for which the Poisson equation is then solved. There is a class of such techniques that are closely related (all scale like $\mathcal{O}(N \log N)$):

- PME (Darden et. al, J. Chem. Phys. **98**, 10089 (1993).)
  use finite Fourier transform of mesh-based charge density (Lagrange interpolation for charges)
- SPME ("Smooth" PME, Essman et. al, J. Chem. Phys. **103**, 8577 (1995).)
  uses B-splines for charge interpolation
- P^3ME (particle-particle-particle-mesh ewald, Hockney and Eastwood, *Computer Simulation Using Particles* (IOP, 1988).)
  optimizes interpolation with respect to minimizing inter-particle force errors

Ewald Mesh Methods Comparison

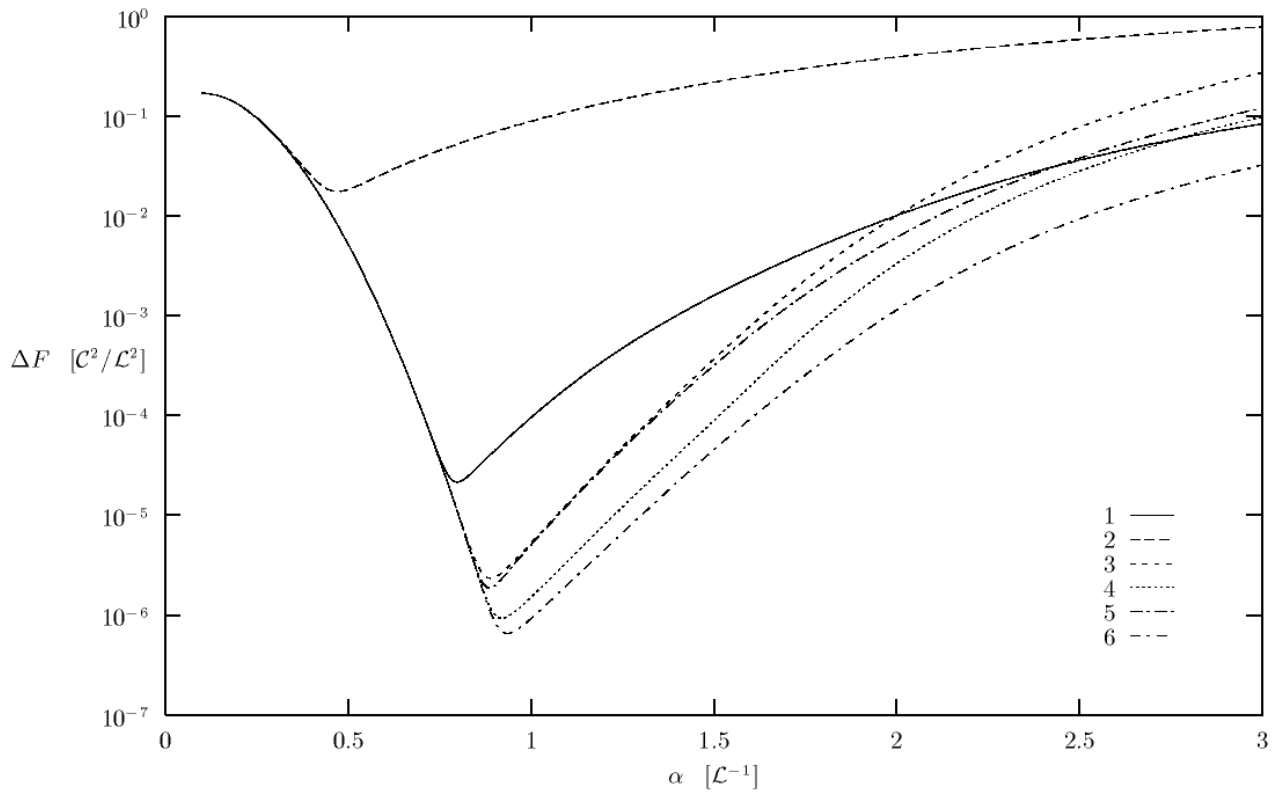


FIG. 2. Comparison of different mesh methods: The rms error ΔF from Eqn. (37) for a system of 100 charged particles randomly distributed within a cubic box of length $L = 10$ (see Appendix D) is shown as a function of the Ewald parameter α for 6 mesh algorithms, which all share $N_M = 32$, $P = 7$ and $r_{max} = 4$. Line 1 is PME. Line 2 corresponds to an algorithm which is obtained from PME by retaining the continuum Green function but changing to the spline charge assignment. Lines 3 and 4 are analytically and $i\mathbf{k}$-differentiated SPME respectively and line 5 and 6 are analytically and $i\mathbf{k}$-differentiated P^3M respectively. Note the logarithmic vertical scale in this and the following figures.

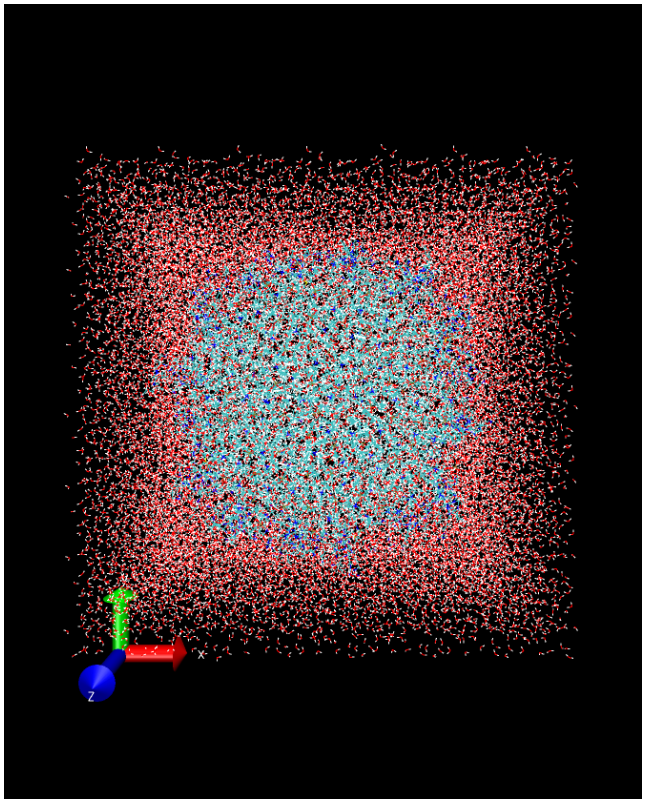
M. Deserno and C. Holm, J. Chem. Phys. **109**, 7678 (1998).

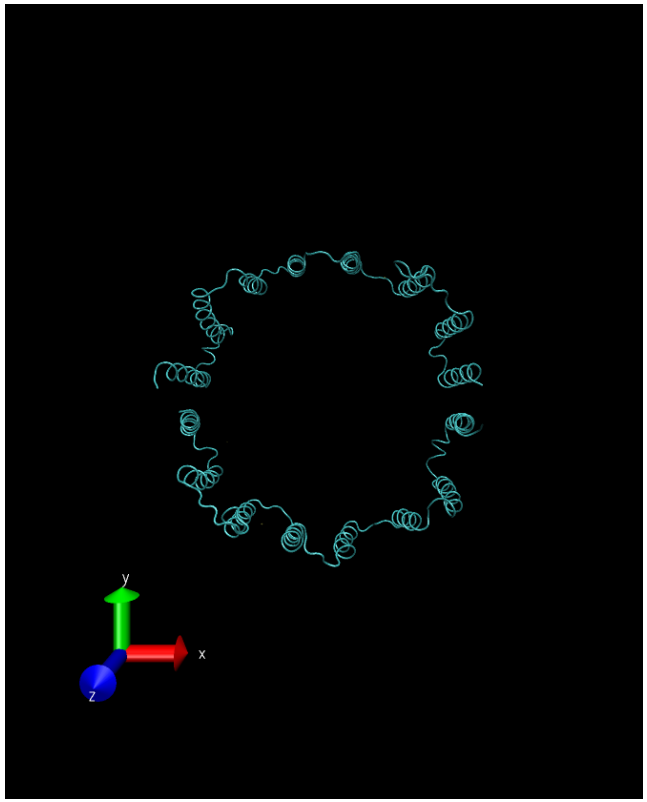
Community Codes that Use Ewald

Some of the dominant players in the world of molecular simulation:

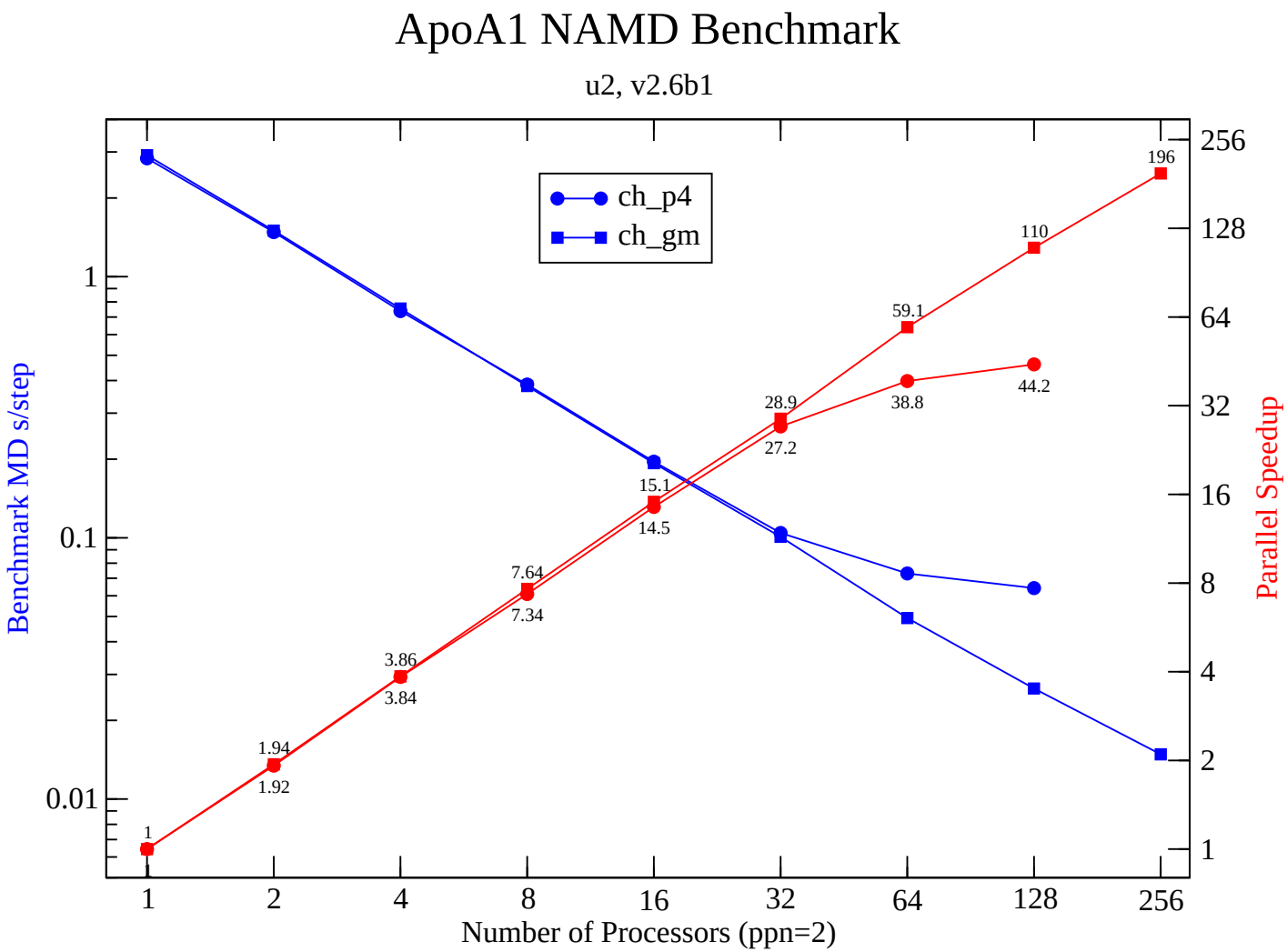
- `CHARMM`, uses Ewald, SPME
  `www.charmm.org`
- `Amber`, Ewald, PME
  `www.amber.scripps.edu`
- `NAMD`, Ewald, PME
  `www.ks.uiuc.edu/Research/namd`

PME/NAMD Example





ApoA1 92224 atoms, 2 proteins, 22010 residues, 21458 waters, PME every 4 steps (12Åcutoff)


ApoA1 NAMD Benchmark
u2, v2.6b1
ch_p4
ch_gm
Benchmark MD s/step
Parallel Speedup
Number of Processors (ppn=2)
1
0.1
0.01
256
128
64
32
16
8
4
2
1
1
2
4
8
16
32
64
128
256
196
110
59.1
44.2
38.8
28.9
27.2
15.1
14.5
7.64
7.34
3.86
3.84
1.94
1.92
1

Barnes-Hut Algorithm

Now we return to the idea of approximation methods to obtain better scalability for very large systems. The Barnes-Hut[3] algorithm is $\mathcal{O}(N \log N)$ in the force calculation algorithm.

The sacrifice in this method (and in all similar methods) is in the details - just like in a multipole expansion. First, though, we need to introduce (or reprise) some details on the fundamental data structures involved, **trees**.

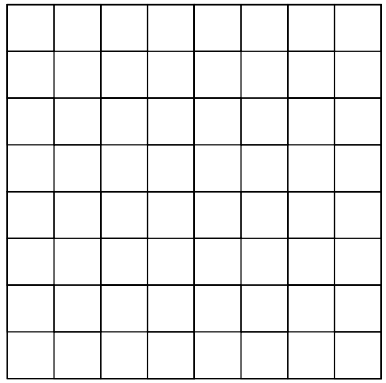
[3] J. Barnes and P. Hut, Nature **324**, 446 (1986)

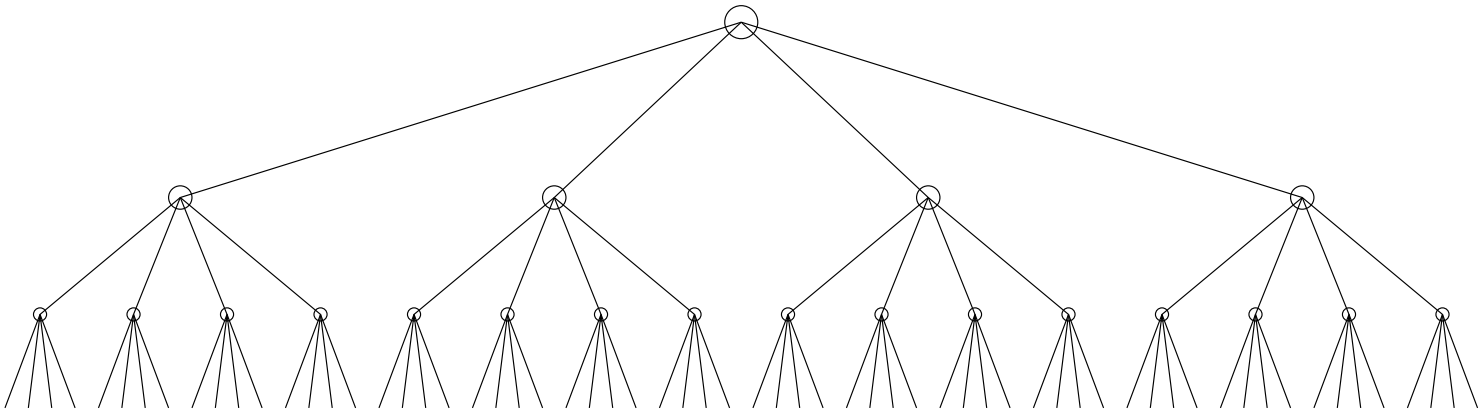
Quad Trees

Quad trees are a data structure to subdivide the plane:

- Nodes can be used to contain coordinates, center of mass, etc.
- A complete quad tree has 4 children in each non-leaf node
- Example of a **complete** quad tree on following slide ...

Complete Quad Tree Example





Oct Tree Example for 3D

Try to draw this one by hand ... ouch!

Back to Barnes-Hut

Ok, so in considering our algorithms we will use trees to hold all the particles, and **adaptive** quad (oct) trees, in which we only subdivide spaces in which particles are located.

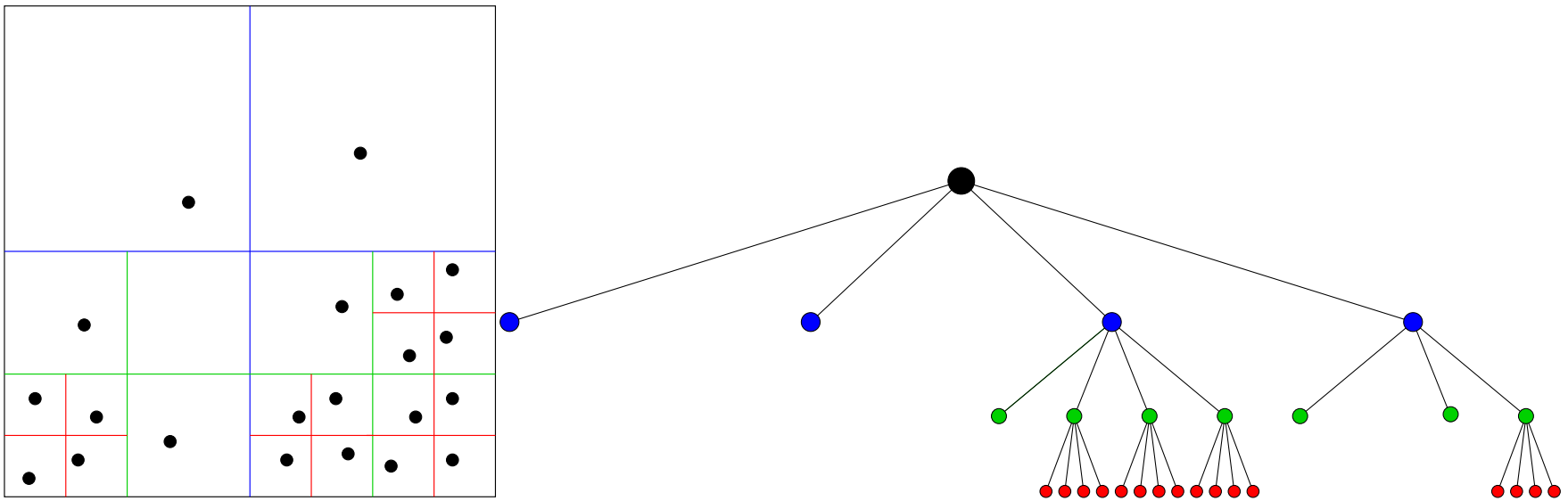
Oct tree algorithm on next slide ...

Hierarchical Decomposition

Here is the rough algorithm describing the spatial decomposition into (in this case an Oct) tree structure:

- Subdivide cube into 8 smaller cubes
- For each cube: if cube does not contain a particle, discard it, else continue subdivision
- Terminate recursive subdivision when cube contains 1 particle. This cube is a **leaf** cube

Adaptive Quad Tree Example



Adaptive Quad Tree Cost

Adaptive Quad tree costs (Oct trees similar):

- For uniform particle distribution: cost is $\mathcal{O}(N \log N)$
- For arbitrary distribution: cost is $\mathcal{O}(bN)$, where b is the number of bits used to store particle positions

Barnes-Hut: Recursive Computation

Barnes-Hut Algorithm:

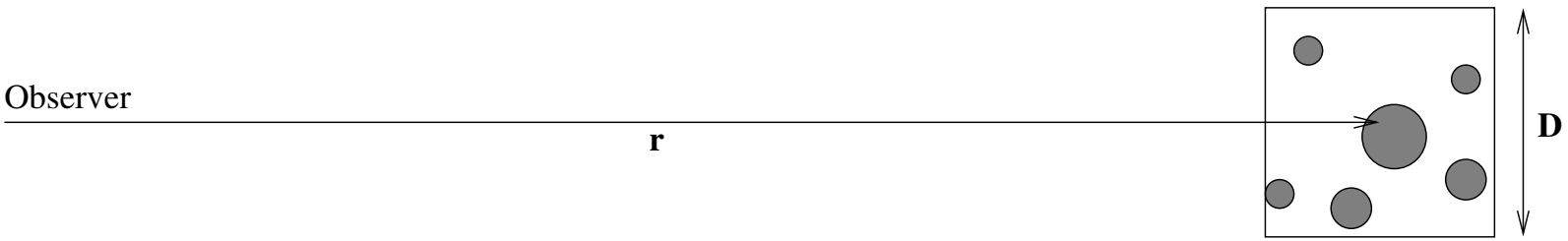
1. Build tree
   - Cost is $\mathcal{O}(N \log N))$ or $\mathcal{O}(bN)$
2. For each node (sub-square or sub-cube) compute center of mass (CoM) and total mass (TM) of all particles contained (recursively)
   - Cost is again $\mathcal{O}(N \log N))$ or $\mathcal{O}(bN)$
3. For each particle, climb/descend tree to compute forces acting on it, using CM and TM of distant sub-squares (cubes)
   - Cost depends on accuracy, but still $\mathcal{O}(N \log N))$ or $\mathcal{O}(bN)$

Barnes-Hut: Force Computation

Force on a particle i due to a node (we know the node's center of mass,

$$\mathbf{f}_i = Gm_i M_{tot} \frac{\mathbf{X}_{cm} - \mathbf{x}_i}{|\mathbf{X}_{cm} - \mathbf{x}_i|^3}$$

this is what is used for "distant" nodes, where $D/r < \theta$ (θ is a user supplied tolerance, and is the primary determinant of accuracy).



Barnes-Hut Accuracy

Accuracy of Barnes-Hut depends on θ, but generally the root mean square error,

$$\texttt{RMS} = \left(\frac{1}{N} \frac{\sum_i ||\tilde{f}_i - f_i||^2}{||f_i||^2} \right)^{1/2},$$

where $\tilde{f}$ is the approximate force, and f the "true" force. This RMS error is typically around 1%.

More Information & Code For Barnes-Hut

- J. Barnes' *Treecode Guide*: http://www.ifa.hawaii.edu/~barnes/treecode/treeguide.html

Fast Multipole Method

The Fast Multipole Method (FMM):

- V. Rokhlin, "Rapid Solution of Integral Equations of Classical Potential Theory" J. Comp. Phys. **60**, 1985
- L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations", J. Comp. Phys. **73** 1987.
- L. Greengard, Ph. D. Thesis, "The Rapid Evaluation of Potential Fields in Particle Systems," Yale 1987 (ACM Distinguished Dissertation Award).

FMM vs. Barnes-Hut

The FMM has some similarities to BH (tree-based, divide and conquer), but also some differences:

- Computes potential, not forces
- Uses more than just the mass information - more expensive, but also (potentially) more accurate
- Fixed set of boxes (no θ parameter)
- BH uses fixed amount of information per box, variable number of boxes (increases with accuracy); FMM uses fixed number of boxes, but data per box grows with increasing accuracy

Kinds of Expansions

Two kinds of expansions in FMM:

1. **Outer** - potential outside node due to particles inside
2. **Inner** - potential inside node due to particles outside (this is the primary goal, of course, for each leaf node)

Multipole Expansion Revisited

Recall our expression for a 3D multipole expansion (this is in Legendre polynomials):

$$V = \sum_{n=0}^{\infty} V_n = \sum_{n=0}^{\infty} \frac{1}{R^{n+1}} \int d^3\mathbf{r}\rho(\mathbf{r})r^n \mathrm{P}_n(\cos\theta).$$

(the following analysis is also valid in 2D) Suppose we keep T terms in the expansion. Then the error in the outer expansion can be upper bounded by $\mathcal{O}(c^{T+1})$. Each term kept adds about 1 bit of accuracy (24 for single precision, 53 for double).

FMM Basic Algorithm

The FMM algorithm can be summarized:

1. Build tree structure
2. Compute outer expansions of each node in tree (traverses tree bottom to top combining expansion of children to get expansion of parent)
3. Compute inner expansions of each node in tree (traverse top to bottom converting outer expansions to inner and combine)
4. For each leaf node, add contributions of nearest particles directly to inner expansion - for one node/leaf, each particle accessed only 1x, $\mathcal{O}(N)$.

BH/FMM Similarities

Similar set of computations:

- Build tree structure
- Traverse tree from leaves to root (outer expansion for FMM, CM/TM for BH)
- Traverse tree from root to leaves (inner expansion for FMM)
- Traverse tree for forces/potential of each particle

Parallelization Scheme

One scheme for BH & FMM:

- Spatial decomposition, shapes adjusted for load balance (each region about N/N_p particles)
- Each process gets tree parts (leaves) containing particles in its region, and parents
- Each process can also store tree parts (limbs?) necessary for computing forces on particles it owns (**locally essential tree (LET)**)

Further Reading

Further references on tree-based N-body methods:

- M. S. Warren and J. K. Salmon, "A Parallel Hashed Oct-Tree N-body Algorithm," Proc. of ACM/IEEE Supercomputing (Portland, OR, 1993), pp. 12-21. http://doi.acm.org/10.1145/169627.169640
- M. S. Warren and J. K. Salmon, "A Portable Parallel Particle Program," Comp. Phys. Comm. **87** 266-290 (1995). http://dx.doi.org/10.1016/0010-4655(94)00177-4
- L. Kale *et. al.*, "NAMD2: Greater Scalability for Parallel Molecular Dynamics," J. Comp. Phys. **151**, 283-312 (1999). http://dx.doi.org/10.1006/jcph.1999.6201