N-Body Pattern Language

Danny Dig (dig@illinois.edu), Ralph Johnson (rjohnson@illinois.edu),
and Marc Snir (snir@illinois.edu)
version May 10, 2009

Abstract

Simulations that depend on interactions between the particles that make up the system are common in in cosmology, molecular dynamics, and computer graphics. When every particle in a system with N particles exercises a force on the other particles in the system, the problems is called N-Body. The naive solution computes forces between each pair of particles (N^2 force computations). This is expensive and does not scale up to large systems. A variety of ways have been developed to solve N-body problems in an efficient and scalable way, leading to O(N log N) or even O(N) time complexity. We present a pattern language for solving N-Body problems that is based on the physical geometry and distribution of the particles.

Introduction

Many simulations contain a set of moving particles. The particles influence each other, but each moves on its own path. The original example was to calculate the paths of "heavenly bodies" like the planets, the sun and the moon. Newton's laws lead to a closed-form solution with two bodies, but the only way to calculate paths of three or more bodies is by simulation, which is why the most common name for this problem is the N-Body Problem.

Newton's model of the solar system was simple; the only force acting on the bodies is gravity. Modern NASA models of the solar system also account for hydrodynamic forces caused by gases near the sun. Other sets of bodies can have even more forces. For example, a set of weights hanging from the ceiling, connected by ropes and springs will have forces due to collision, to springs and ropes, and due to friction from the air. A set of atoms in a living cell will have forces caused by molecular bonds, by electrical charges, and by other forces such as Van der Walls forces. In theory, all these systems can be simulated by the same algorithm. In practice, as system size grows, more specialized algorithms are needed.

The fundamental algorithm for simulating a particle system is to keep track of the location and the velocity of each particle, as well as other information like mass and charge. To simulate how the particles move, first pick a time duration dt, calculate the force on each particle, and then calculate the new velocity of each particle by adding in the effect of the force (force * dt) and the new position of each particle by adding in the effect of the velocity (velocity * dt). Repeat.

```
for t = 0 to t_final step dt
    for i = 1 to n
        compute force(i) = force on particle i
    end for
    for i = 1 to n          ... n = number_of_particles
         velocity(i) = velocity(i) + force(i)/mass(i)*dt
         position(i) = position(i) + velocity(i)*dt
    end for
    compute interesting properties of particles
end for
```

There are several possible sources of error in this simulation. First, the computation of force(i) might have some error. There are many ways to compute the force, and some trade accuracy for speed.

Another is that velocity and force are not constant during the entire duration dt. The difference between the velocity of a particle at time t and at time t+dt depends on its acceleration, i.e. on the forces on it. Since the force computation is usually the most expensive part of the algorithm, an easy way to reduce error is to break up dt into smaller time intervals and calculate the velocity several times for each force computation. For example, if dt is split in two then we would calculate the velocity twice for each force computation, as follows:

```
for t = 0 to t_final step dt*2
    for i = 1 to n
        compute force(i) = force on particle i
    end for
    for i = 1 to n
         velocity(i) = velocity(i) + force(i)/mass(i)*dt
         position(i) = position(i) + velocity(i)*dt
    end for
    for i = 1 to n
         velocity(i) = velocity(i) + force(i)/mass(i)*dt
         position(i) = position(i) + velocity(i)*dt
    end for
    compute interesting properties of particles
end for
```

The duration dt should be small relative to the speeds of the particles. This leads to many time steps, since dt is often a thousandth or a millionth or even a billionth of the total simulation time. The smaller the duration, the more accurate the simulation, but the bigger the duration, the faster the simulation. One way to improve the balance between accuracy and speed is to vary the duration, making the duration smaller when when the forces get larger (which usually means that the particles are close together) and making the duration larger when the forces are smaller. This is described by the first, pattern *Variable Time-steps*.

Computing the force takes up most of the CPU cycles for simulations, often over 90%. This is because computing the force on a particle often requires examining every other particle. For example, each particle has a gravitational attraction to every other. Thus, computing the force on a particle can take time linear to the number of particles, and the algorithm for solving the N Body problem will take time $O(N^2)$. This will not scale for large systems. So, N-Body simulations for large number of particles will either group particles and treat their contribution to the force as coming from a single particle, or will generate a force field and then calculate (in linear time) the force on each particle. These alternative methods always lose a little accuracy, but different methods have different weaknesses so it is usually possible to choose a method that fits the problem well. This paper describes four patterns that involve different ways of computing forces, *Particle-Partical, Tree-code Top Down, Tree-code Bottom Up,* and *Particle-Mesh.*

The force on a particle often has many components. Part of the force might be a function of every other particle, but part might not depend on the other particles, or only depend on a few of the particles. Since forces are additive, it is often useful to *Split Forces* and compute different components of the force differently. Parts of the force that do not depend on every other particle can usually be computed in linear time, and can be parallelized with *Task Parallelism* or *Geometric Data Decomposition*.

The following six patterns make up a pattern language for solving N-Body problems. The first two patterns are *Variable Time-steps* and *Split Forces*; the next four are about various ways of computing forces.

The Pattern Language

Variable Time-steps

What:
Instead of using a constant time-step, use a large time-step when forces are large, and a small time-step when forces are small.

Use When:
- collision is possible, and must be modeled accurately
- it is easy to divide particles into those that are close and those that are far

How:

When particles get close together, the force between them can be strong. Moreover, as one particle passes close by another, the force can change rapidly. This case requires a small time step. So, one way to make a simulation both fast and accurate is to vary the time step. Use a large time step whenever particles are not close together and a small time step when they are. Another way of saying this is to use a large time step when forces are small and not changing quickly, and a small time step when forces are large and changing quickly. This is especially useful when collision is rare but must be modeled accurately.

In fact, it is possible for each particle to have a time step different from every other particle. Suppose that there is a sequence of sets of particles, where particles in S_{i+1} have time steps half the size of the time steps of particles in S_i. In general, the simulation would perform two time steps for each particle in S_{i+1} before performing a time step for particles in S_i. If the force on a particle was too large then it would be moved to the set of particles with a smaller time step, and if the force was smaller than a threshold then it would be moved to the set of particles with a larger time step.

However, the most common way to vary time steps is not based on the force, but by distance. A common way to speed up a N-body problem is to divide the space into regions, to compute the total force caused by the region, assuming that the region was represented by a single particle. This will not be accurate close to the particle, but it will be accurate far away. So, the force on a particle will be computed by examining every particle in nearby regions, but by using only the representative particle for other regions. If N particles are evenly divided through space (which is not true for astronomical systems, but is true for atomic models of metals) then dividing space into $\sqrt{N}$ regions each with $\sqrt{N}$ particles will lead to an algorithm that is $O(N\sqrt{N})$. Instead of calculating the representative particle each time step, it could be computed every tenth time step, or every hundredth. Morover, since

the representive is far away, the force contributed by distant regions could be contributed less frequently, as well.

Related:

Vareiable Time-steps work well with *Particle-Particle, Tree-code top down* and *Tree-code bottom up*, but doesn't work well with *Particle-Mesh.*

Split Forces

What:
Compute the total force on a particle by separately computing each kind of force and then adding them together.

Use When:
- forces are complex, with many components
- some of the component forces are easy to compute

How:

Since the total force on a particle is the sum of the forces from each of the other particles, the total force does not have to be computed all at once. Sometimes there is an efficient way to compute one of the contributing forces, and then it makes sense to compute it individually.

For example, in a molecular dynamics simulation, one of the most important forces on an atom is an atomic bond. However, each atom is in a small number of atomic bonds that usually do not change during the course of the simulation. So, give each atom a list of its atomic bonds. The amount of time to compute the forces due to bonds is proportional to the number of bonds, not the number of particles in the system. Thus, the total time to compute the forces due to bonds is linear, and it makes sense to compute them separately.

Van der Waals forces are inverse sixth, and so distant particles can be ignored. If the overall simulation has a data-structure that provides geometrical information then it is best to compute van der Waals forces separately from inverse squared forces like electrical charge.

Related:

Particle-Particle (PP)

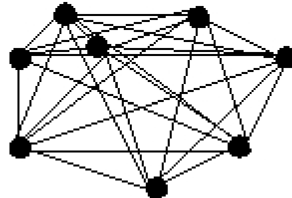
What: Compute the forces as the pair-wise interaction between all particles in the system.

Use When:
- the number of particles is small (< 1000)

- collisions or near-misses are important
- accuracy is very important
- number of particles is large but one has special purpose hardware (e.g., GRAPE computers)

How:

Particle-Particle is the most straighforward solution to the N-body problem. It treats all particles equally and it computes the forces between all pairs of particles (see Fig 1).



For example, in a gravitational N-body simulation, a particle of mass M_1 attracts another particle of mass M_2 with a force:

$F= - G *M_1*M_2 / r^2$ (where *G* is the gravitational constant and *r* is the distance between the particles)

To compute the force on a particle *i*, one needs to accumulate all forces F(i,j) of particle *j* on particle *i*.

Such a computation is highly accurate, but fairly ineficient since it has a computational complexity $\mathbf{O(n^2)}$ where *n* is the number of particles. However, it is easy to parallelize. In fact, it is possible to build special purpose hardware to achieve high performance. The GRAPE (GRAvity PipE) systems consist of a ring of processors that use pipeline parallelism [Makino], and have won the Gordon Bell award seven times between 1995 and 2003.

Related:

All the patterns in the language use different approximations to cut down the $\mathbf{O(n^2)}$ cost of the Particle-Particle simulations. Also, rather than using a constant time-step, there are several methods that have a variable time-step integration (e.g., Bulirsch-Stoer method).

Particle-Mesh (PM)

What:
Use a mesh to generate a force field generated by a set of particles. After computing the force for the entire field, interpolate the force on the individual particles.

Use When:
- particles are uniformly distributed (e.g. as in liquid molecular dynamics)

- collision can be ignored

How:

Start with the density function. Locations with particles have a density based on the mass of the particles, but locations without particles have zero density. Discretize space by dividing it into a uniform mesh. Assign each particle to a point or set of neighboring points on the mesh. The mesh should be fine enough that no point has more than one particle assigned to it.

There are a variety of ways of assigning density to mesh points. The best methods will apportion the mass of a particle to several grid points, thus treating particles more like fuzzy balls than points.

To convert the density function into a potential field,

1) do a 3-D FFT on the density.

2) convolve the results with the sum of inverse square frequencies. This computes the potential in the frequency domain.

3) do an inverse 3D FFT back to the original mesh points, producing the potential field.

Force is the negative of the gradient of the potential field. So, compute the force at each point on the mesh.

The algorithm has complexity O(M ln M) where M is the number of mesh points. However, if particles are not evenly distributed, the number of mesh points might be much larger than the number of particles.

Related:

The main weakness of a PM method is that it is not accurate for forces between particles that are close to each other. Particle-Particle/Particle-Mesh solves this main weakness by splitting the forces into a rapidly-varying, short range part and a slowly-varying, long range part. The Particle-Particle pattern is used to find the short-range contribution to the force and Particle-Mesh is used to find the total long-range contribution. Particle Mesh Ewalt is one of the most popular of the Particle-Particle/Particle-Mesh algorithms, and is used by NAMD[PME][NAMD]. One of the advantages of Particle Mesh Ewalt is that it increases locality and so is more suited to a distributed memory system than Particle-Particle[Snir 2004].

A Tree-Code Particle Mesh is a hybrid of Tree Codes with Meshes. It uses a Tree Code approach in regions where the mass density is high and a Particle-Mesh approach in other areas.

:

Tree-Code (TC) Top Down

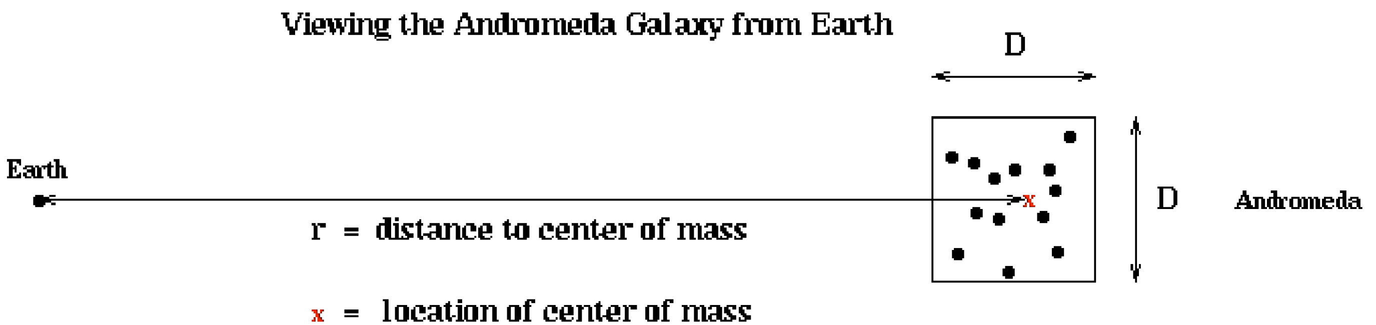
What:
Tree-Code is a divide-and-conquer technique: it decomposes the space into a hierarchical tree. When computing the force on a given particle, the tree-region near the particle is explored in detail while the more distant regions are explored more coarsely by treating distant clumps of particles as one single massive pseudo-particle.

Use When:

- particles are unevenly distributed, as in cosmological simulations

How:

The intuition is that as a clump of particles get much further away from the particle of interest, they can be aproximated with one single massive particle located at the center of mass of the clump and having a mass equal with the total mass of the clump. For instance, Andromeda galaxy appears to the naked eye as one single star even though it contains billions of stars (see Fig Andromeda):



If the ratio D/r = (size of box containing particle) / (distance of center of mass from the particle) is small, it is safe to replace the whole clump of particles with one single particle. The same ideas applies recursively within the box, as long as the ratio D/r is small, the particles in the smaller box can be replaced by their center of mass.

There are a variety of algorithms that are top-down tree-codes. One of the most popular is Barnes-

Hutt, which has three steps:

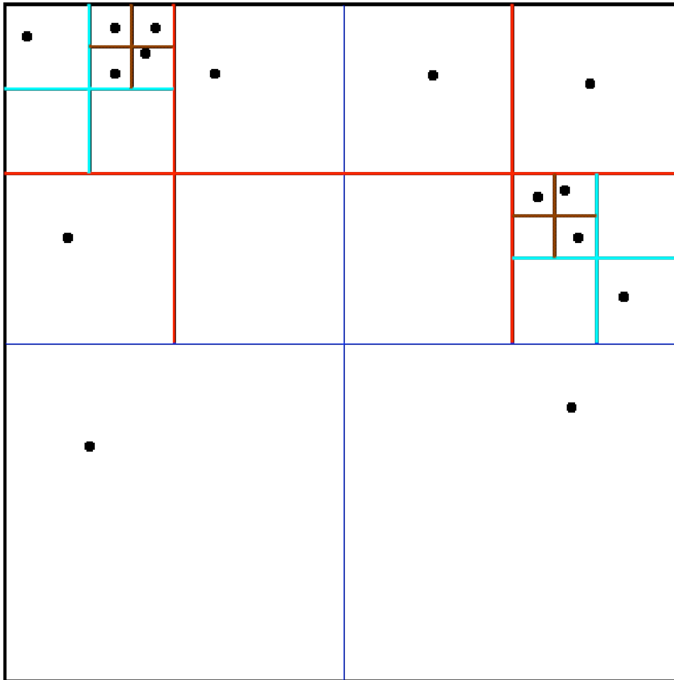
1. Build a quadtree/ octtree depending on whether the problem is 2D or 3D.
2. For each node in the tree, compute the center of mass and total mass for all the particles it contains
3. For each particle, traverse the tree to compute the force on it.

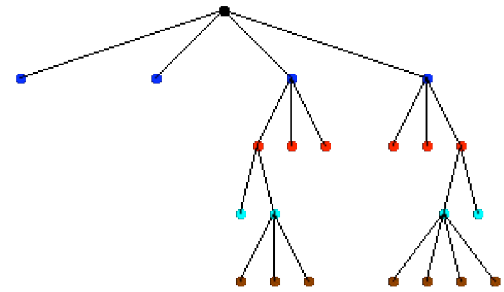
Step 1: Build a quadtree

A data structure that allows us to subdivide the space recursively is quadtree (for 2D) or octtree (for 3D). Here we describe the quadtree, but the octtree is analogous (just add one more dimension). The quadtree begins with a square in the plane. This large square is divided into four equal smaller squares. Each child square can be further divided into four smaller squares and so on. The division can continue until each square contains only one single particle. The algorithm builds a tree based on the spacial decomposition: each internal node in the tree corresponds to a square, each leave of the tree represents a particle.

When the particles are non-uniform distributed, it does not make sense to further divide the squares that have no particles. Therefore, their equivalent nodes in the quadtree would be empty. Figure Quadtree shows a tree with five levels, children are ordered counter-clockwise starting at the lower left corner (notice that the color of the nodes in the quadtree is the same as the square that immediately contains the particle.

Adaptive quadtree where no square contains more than 1 particle





Step 2:

For each node in the tree, compute the center of mass and total mass for all the particles residing in the corresponding square. This is accomplished by a simple *post-order* traversal of the tree. The data is computed only once and stored in the quadtree.

Step 3 :

For each particle, traverse the tree to compute the force on it. We show the 2D computation, a similar approach is used for 3D.

If D(size of square)/r(distance from particle to a square) is smaller than a user threshold, _, compute the force on a particle as follows. Let:

- (x, y) be the position of particle
- m be the mass of the particle
- (x_{CM}, y_{CM}) be the position of the center of mass of particles in the square
- m_{CM} be the total mass of particles in a square
- G be the gravitational constant
- r is the distance from the particle to the center of mass of particles in the square
  $r = sqrt((x_{CM}-x)^2 + (y_{CM}-y)^2)$

$$force(particle, square) = G*m*m_{CM}*((x_{CM}-x)/r^2 , (y_{CM}-y)/r^2) \quad (*)$$

```
//for each particle, traverse the tree to compute the force on it
for i = 1 to N
        f(i) = TreeForce (i, root)
end for

TreeForce(i, node)
        f = 0
        if node contains one particle
                f = force computed using formula above (*)
        else
                if D/r <  _
                        compute f using formula (*) above
                else
                        for all children c of node
                                f = f + TreeForce(i, c)
                        end for
                end if
        end if
```

The algorithm complexity for computing force is O(N * log N) where N is the number of particles. The downside is that tree codes require a large amount of storage data.

Discussion:

Related:
One of the advantages of tree codes is that they have no prefered geometry and adapt to the geometry of the problem. They work especially well for cosmological problems where they focus on the small fraction of space that contains matter. They are not used as often for molecular dynamics, where distribution of particles is uniform.

This algorithm was presented in "A Hierarchical O(n log n) force calculation algorithm" by J Barnes and P.Hut, Nature, v324, December 1986, and is based on an earlier algorithm of A. Appel in 1985.

The Fast Multipole Method (FMM) is a more accurate, and more complex, tree-code method. The center of gravity is only a good approximation to a potential field if all the particles are far away. It is just the first term in the multipole expansion of the potential. FMM keeps track of other terms, as well, so Barnes-Hutt is essentially a dumb FMM with order-0 multipole expansions. The FMM has many advantages; under certain circumstances it is O(N) and it allows a precise measure of the accuracy of its result, which is not possible even with particle-particle methods. However, the FMM is complex, and is slower than O(N log(N)) methods even for systems of hundreds of thousands of particles, so it is not used often.

Tree-Code (TC) Bottom Up

What:

Like in the Tree-Code Top Down, this pattern decomposes the space into a hierarchical tree. However, the tree is built from the bottom up. This is sometimes referred as the "Press Tree".

Use When:

- one wants to give individual timesteps to each of the particles with smaller timesteps in sections where interesting close encounters are occuring
- one wants to handle close distances correctly

How:

Build the tree from the bottom. First, find the two particles nearest each other and make them leaf nodes of a tree of size three, with the root of the tree being their center of mass. Do the same with the next two particles that are nearest each other. Keep pairing particles until there are less than two particles left. Then repeat the process, but now with the roots of trees from the previous phase. The root of each new tree will be the center of mass of its two subtrees. Continue until there is only one tree.

Once the tree is formed, compute the force on each particle by traversing the force tree.

Once the particles move, the tree will have to be recomputed. In general, the structure of the tree will not change that much from one iteration to the next. Nevertheless, sometimes the nearest neighbor of a particle in a new tree can be very far from it in the old tree.

It is easy to have *Variable Timesteps.* Nodes in the tree should have timesteps at least as big as any of their children. It is best to make every timestep be some power of two times the smallest timestep.

The computational complexity is O (N logN)

Related:

While it has greater flexibility (using different timesteps) than the Top-Down pattern, this pattern is also more complex to understand and code.

To do:

The Symplectic Method.

References

[Gibbon & Sutmann] P. Gibbon and Sutmann, G. Long-Range Interactions in Many-Particle Simulations. Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms, Lecture Notes, J. Grotendorst, D. Marx, A. Muramatsu (Eds.),John von Neumann Institute for

Computing, Julich, NIC Series, Vol. 10, ISBN 3-00-009057-6, pp. 467-506, 2002.

[Makino] Junichiro Makino and Makoto Taiji, Scientific Simulations with Special-Purpose Computers--*the GRAPE Systems*, John Wiley & Sons, 1998

[Phillips et. al] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kale. NAMD: Biomolecular simulation on thousands of processors. In Proceedings of Supercomputing, The SCxy Conference series, Baltimore, Maryland, November 2002. ACM/IEEE.

[Snir] M. Snir. A Note on N-Body Computation with Cutoffs. Theory of Computing Systems 37, 295-318 (2004) Springer-Verlag.

[Toukmaji, Paul and Board] Abulnour Toukmaji, Daniel Paul, and John A. Board Jr. Distributed Particle-Mesh Ewald: A Parallel Ewald Summation Method. Tech report 96-003, Duke University Department of Electrical and Computer Engineering. Appeared in Parallel and Distributed Processing Techniques and Applications Conference, Aug 9-11, 1996, Sunnyvale CA.