Parallel Tree Codes

Introduction

During the past decade, N-body tree codes have been applied successfully to various problems in galaxy dynamics, galaxy formation and cosmological structure formation. Although the number of distinct particle-pairs in an N particle system is proportional to N^2 for large N so that direct methods scale as $O(N^2)$, the computing time for N-body tree code implementations only scales as $O(N \ell og N)$. This means they provide a relatively fast way to solve the general collisionless N-body problem. At this point in time, however, tree code simulations running on fast workstations and vector supercomputers have been restricted to $N < 10^6$ because of memory and time limitations.

It is very important to increase the size and speed of the simulations since the discreteness of N-body systems can lead to false relaxation and heating of collisionless systems like galaxies on short timescales and mask real dynamical effects. In addition, fine scale features like shells and tails which develop in when galaxies interact only become apparent in simulations when N is large.

Also during the past decade, there has been a paradigm shift in supercomputing with the movement from single vector (special coding) machines to the new massively parallel machines and clusters. Parallel computers or clusters are collections of independent smaller processors interconnected with an internal high speed communications network. The AppleSeed cluster that exists in the basement of DuPont during the summer months is an example (12-16 nodes on a 100-megabit ethernet backbone using MPI-Pooch software from UCLA/Dauger Research). In a parallel cluster, each processor operates independently but communication software allows messages containing data to be exchanged rapidly with other processors. In principle, a problem can be partitioned among the N processors and a maximum N-fold increase in computational speed can be realized. In practice, however, the speed up is smaller because of the extra time needed for exchanging data in message-passing algorithms.

Parallel machines or clusters offer a new route to very fast computation only if algorithms can be redesigned to conform to the message-passing paradigm and communication can be minimized. These new algorithms are very different from their sequential counterparts and generally require a considerable effort to redesign. The key to a successful algorithm is good **load balance**, i.e., where both **data** and **computational work** are distributed evenly among the processors.

A common way of achieving load balance on parallel machines or clusters is through domain decomposition. The physical domain of the problem is partitioned into smaller subdomains and the physical quantities of these subdomains such as values of density, pressure and temperature at grid elements in a fluid code or collections of particles and their attributes in an N-body code are assigned to each processor. The most important part to designing a parallel algorithm is finding a way of partitioning the domain so that data and work are evenly distributed (load balance) and communication is minimized.

In this research, we will be investigating several different parallel tree algorithms with the goal of producing a more robust and stable implementation and developing a good starting point for other parallel algorithm development in N-body dynamics.

Tree Codes

The N-body problem involves advancing the trajectories of N particles according to their time evolving mutual gravitational field. In the simplest algorithm, the force on each particle is determined by direct summation of the contributions from the other $N-1$ particles. In a discrete time integration, the forces at each timestep are then used to advance the particles along their trajectories according to a numerical schemes such as the leapfrog method (Euler-Cromer). Computational costs of direct summation scale as $O(N^2)$, making the algorithm expensive. In collisionless systems like galaxies, however, a simulation can tolerate small errors in the force for improved performance by using techniques which approximate the gravitational field. Tree codes are one class of methods which accomplish this and have the advantage of scaling only as $O(N\ell ogN)$ in computational cost.

The essence of a tree code is the recognition that the gravitational potential of a distant group of particles can be well-approximated by a low-order multipole expansion. In a tree code, a set of particles is arranged in a hierarchical system of groups that form of a tree structure. The entire set is subdivided into several groups and each of these groups is broken down in succession within the hierarchy until groups contain at most 1 particle. There are many different methods available for efficiently grouping particles hierarchically in this way.

The evaluation of the potential at a point reduces to a descent through the tree. One sets a minimum distance a point can be from a group to use a multipole expansion. If the point is sufficiently distant from a group, the multipole expansion is used to find the potential from that group (much faster than direct summation methods), while if the point is too close to the group, each of its child subgroups are examined. The procedure continues until all groups are broken down as far as they need be. Some particle that are close to the point will be groups by themselves and for those we still use direct summation. This et, however, will be very small in number.

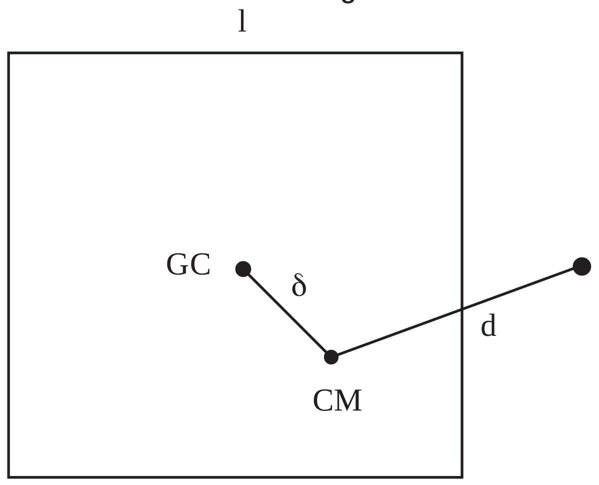
The Barnes-Hut Tree Code

The Barnes-Hut algorithm works by grouping particles using a hierarchy of cubes arranged in an oct-tree structure, i.e., each node in the tree has 8 siblings. The system is first surrounded by a single cube or cell encompassing all the particles. This main cell is subdivided into 8 subcells, each containing their own subset of the particles. The tree structure continues down in scale until cells contain only 1 particle. For each cell or node in the tree, we calculate the total mass, center of mass and higher order multipole moments (typically only up to quadrupole order). This tree structure can be built very rapidly (only takes a few percent of the total time per step) making it feasible to rebuild it at each time step.

The force on a particle in the system can be evaluated by **"walking"** down the tree level by level beginning with he top cell. At each level, a cell is added to an interaction list if the cell is distant enough for a force evaluation. If the cell is too close, it is "opened" and the 8 subcells are either used for force evaluation or opened further. The walk ends when all cells which pass the opening test and any remaining single particles are acquired. The accumulated list of interacting cells and particles is then looped through to calculate the force on the given particle and this amounts to the bulk of the computation. In this way, the number of interactions computed is significantly smaller than a direct N-body method. Typically, there are only ~ 1000 interactions per particle on average in a simulation with 10^6 particles making the algorithm significantly faster than the direct summation method.

Cell-Opening Criterion

A cell-opening criterion is used to determinine whether a cell is sufficiently distant for a force evaluation via multipoles. The simplest criterion is based on an opening angle parameter θ. We consider the cell shown in the figure below



Geometry of the Barnes cell-opening criterion used in the parallel tree code

where GC = geometric center of the cell and CM = center of mass of the cell. If the size of the cell is l and the distance of the particle from the cell center of mass is d, the algorithm accepts the cell for a force evaluation if

$$d > \frac{l}{\theta} \qquad (01)$$

Smaller values of θ lead to more cell openings and more accurate forces. Typically, $\theta = 1$ gives accelerations with errors around 1%.

It turns out, however, that this criterion for calculating the potential causes gross errors in some pathological cases in which the CM is near the edge of the cell. A slightly different criterion (determined by trial and error) for cell opening

$$d > \frac{l}{\theta} + \delta \qquad (02)$$

where δ is the distance between the CM and the GC circumvents this difficulty. This criterion guarantees that if the CM is near a cell edge, only positions removed by an extra δ use the cell for a force evaluation, while if the CM is near the cell center it reverts to the old criterion.

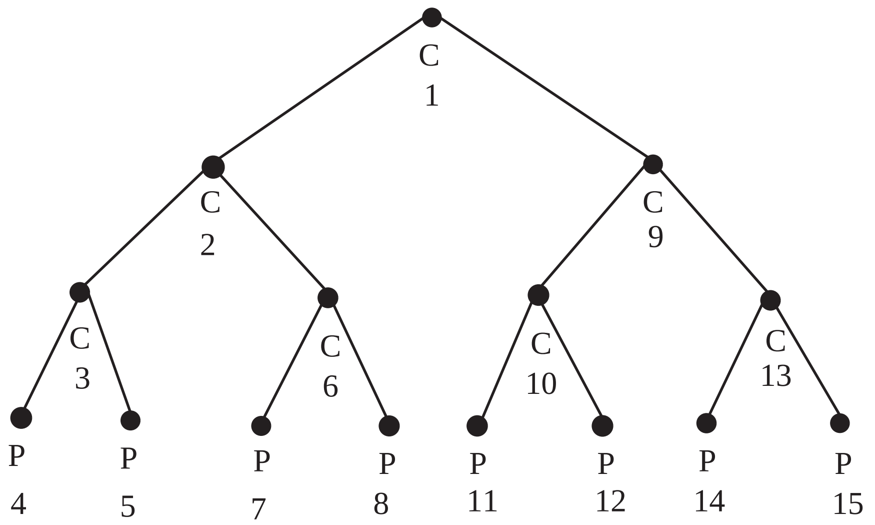
Grouping

These tree walks mean a sizable overhead in computation since the cell opening criterion must be computed many times for each particle. A slightly different strategy reduces the number of tree walks for a net gain in computing speed. The strategy is to find all cells in the tree which contain less than $N_{crit}(\sim 32)$ particles. The tree walk proceeds as before but instead of defining d as the particle distance from a target cell's CM, d is defined as the distance between the nearest edge of the cube encompassing the group to the target's CM. The accumulated interaction list is then used for all the particles in a given group and the forces are guaranteed to be at least as accurate as those of an individual particle at the edge of the group's cube. This leads, in practice, to a considerable speedup for fixed force accuracy.

Non-Recursive Tree Walks

When tree structures are part of an algorithm, it might seem natural to program functions recursively. Unfortunately, there is a large overhead from any recursive calls. The tree walk is called many times at each time step and so it is best to eliminate any recursion in this function. Non-recursive tree walks can be accomplished by arranging the tree nodes in a **linked list**. Each node is set up to contain a pointer to its first child and adjacent sibling. If a node is the last child in a level, its sibling pointer is assigned to its parent's sibling. Thus, a node which contains 1 particle will have only one adjacent sibling and no children.

The nodes are resorted as follows (see figure below). The first node



Idealized binary tree showing the order of nodes for a non-recursive tree walk. Cell nodes are labelled with C and particle nodes with P. The index of the node in the list is also shown.

in the list is set to the root cell containing all of the particles.

The descent of the tree always proceeds to the current node's first child which is then added to the list. If the node contains only one particle the walk proceeds to the node's sibling which is then added to the list. Once the nodes are sorted this way, a tree walk for a force calculation then reduces to a scanning of the list. If a cell must be opened, one simply examines the next node in this list which is guaranteed to be the first child cell at the next level of the hierarchy. If a cell can be used, the walk to the sibling is simply a jump down the list to the appropriate cell labelled by the sibling pointer. The other main advantage of resorting the tree nodes this way is that it becomes easy to prune and graft trees onto existing structures which will be essential for the parallelization of the code.

This tree algorithm needs to be implemented on each processor in the parallel cluster. Each node level code is optimized using grouping and non-recursive tree walks which improve the tree code's efficiency.

Parallel Tree Codes

The path to parallelization of the BH algorithm is not obvious since the inhomogeneous distribution of particles in the oct-tree structure does not immediately lend itself to a simple decomposition with load balance. The main difficulty is that both the particles and the tree structure must somehow be distributed in a balanced way among many independent processors. The is especially true on a loose network of workstations each of which have relatively small amounts of memory (like our AppleSeed cluster).

One possible parallelization of the tree code, which retains most of the features of the original tree code on the level of individual processors but introduces a new algorithm for distributing the particles and tree structure among the processors, is orthogonal recursive bisection.

Orthogonal Recursive Bisection

In the N-body problem, the system of particles is enclosed by a rectangular box for isolated systems like individual galaxies or an exact cube in cosmological systems with periodic boundaries. We now discuss a parallel algorithm for decomposing this volume into rectangular sub-volumes using the method of orthogonal recursive bisection (ORB).

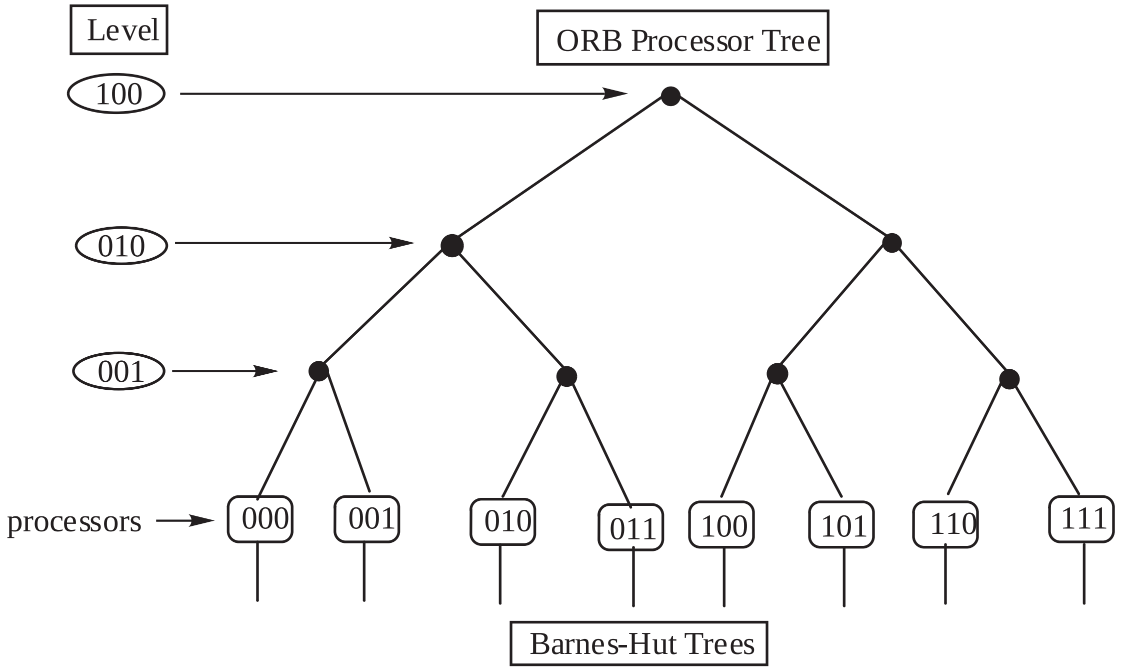
In this technique, a volume is represented as a hierarchy of rectangular boxes somewhat like the BH tree. The main volume is first cut along an arbitrary dimension at an arbitrary position. For a balanced tree, the position is chosen so there are equal numbers of particles on each side of the cut. The resulting two volumes are cut again along the same or a different dimension again at an arbitrary position. The slicing of each subvolume continues for as many levels as required. The resulting process can be represented by a binary tree structure since each node points to two children. An ORB tree of n levels therefore results in 2^n subvolumes.

At each level in the construction of the ORB tree, we have the freedom to choose both the dimension and the position of the volume subdivision. For example, if the initial volume is a cube, one can construct an ORB tree analogue of the BH tree by cycling through the 3 dimensions and cutting only through the geometric center of each subvolume. With load balance in mind, however, we wish to position our slice of the subvolume so that there are equal amounts of **computational work** on each side of the slice. This trick will achieve load balance. We keep track of the computational work for an N-body simulation by simply summing up the number of cell-particle interactions, the main sink of computing time. In this way, once the particles in each subvolume are assigned to a processor, the calculation of the acceleration should be load balanced. Initially, the amount of computational work is unknown so it is assumed to be the same for each particle. The first step is therefore somewhat load imbalanced but successive steps become more balanced.

The choice of dimension for cutting the subvolumes in the ORB tree construction is generally arbitrary but it is best to select the longest dimension for slicing. The tree code uses a multipole expansion to quadrupole order so we desire rectangular domains that are as **"spherical"** as possible so that higher order multipoles contributions are negligible.

Linking the Trees

After the decomposition, a BH tree is constructed locally in each processor using the particles contained within its independent subvolume. After constructing local trees, we would ideally link them together to form the full tree describing the entire system. The top levels of the tree are simply the load balanced ORB tree structure created by the domain decomposition, while the processors contain the BH trees (see figure below)



The hybrid tree describing the hierarchical grouping of the particles used for the tree walk. The top levels of the tree are the binary structure formed by the ORB domain decomposition. The BH trees built in each processor are grafted to the leaves of this tree to form the hybrid. Each processor contains only a pruned version of the other processor's BH trees.

If a copy of this complete tree structure were available to each processor, they could proceed independently to evaluate forces on their particles. Unfortunately, the amount of memory required for a full copy of the tree at each node is prohibitive.

The problem is solved by introducing the idea of a **locally essential tree**. Each processor does not require an entire copy of the tree. Rather, only a significantly smaller subset of nodes is necessary since the BH trees in distant volumes need only be opened to a lesser degree for **all** the particles in a given processor domain. The opening criterion can be applied to the entire group of particles in a processor by calculating the distance to the nearest edge of a processor volume in the same way as in the grouping scheme used to improve the efficiency of the force evaluations. Therefore, a processor only needs to import significantly pruned trees from distant processors which it can graft onto its existing structure to create the locally essential tree. The pruned tree of the entire system contains all the nodes required to calculate the forces to the required tolerance specified by the opening angle criterion.

The locally essential trees are constructed in the following manner.

After building the local BH-tree, each processor imports the root nodes of the trees from all the other processors. A local binary tree is built from the base up in each processor using the imported root nodes. A walk through the ORB tree using a group opening criterion determines which subset of processor must be examined further to gather more tree nodes if necessary. Tree walks are performed in the needed processors using the group opening criterion of the requesting processors. BH tree nodes which can be used are flagged. The nodes of the pruned trees are then gathered together and exported to the calling processor. Once received the internal child and sibling links of the imported trees are reset and the pruned trees are grafted at the appropriate node of the ORB binary tree. The result of this procedure is the locally essential tree which contains: the local BH-tree, the ORB tree structure and the trees pruned to the appropriate levels imported from other processors.

Summary

The parallel tree code works as follows. At the start, particles are distributed randomly among the processors. At each time step, the following procedures are carried out:

(1) ORB domain decomposition across processors
(2) Construct the local BH tree
(3) Exchange tree nodes to construct the locally essential trees
(4) Walk through the trees to calculate the forces
(5) Move particles along their trajectories using these forces

Only steps (1) and (3) require message passing. In both cases, messages are exchanged in a synchronous manner using the message-passing interface (MPI). The messages contain arrays of particle data (masses, positions, and velocities) and tree node data (masses, positions, quadrupole moments and cell dimensions).