A Portable Parallel Particle Program

Michael S. Warren
Fluid Dynamics, T–3
Mail Stop B216
Los Alamos National Laboratory
Los Alamos, NM 87545

John K. Salmon
Physics Department
206-49
California Institute of Technology
Pasadena, CA 91125

October 24, 1994

Abstract

We describe our implementation of the parallel hashed oct-tree (HOT) code, and in particular its application to neighbor finding in a smoothed particle hydrodynamics (SPH) code. We also review the error bounds on the multipole approximations involved in treecodes, and extend them to include general cell-cell interactions. Performance of the program on a variety of problems (including gravity, SPH, vortex method and panel method) is measured on several parallel and sequential machines.

1 Introduction

There are two strategies that can be applied in the quest for more knowledge from bigger and better particle simulations. One can use the brute force approach; simple algorithms on bigger and faster machines (and bigger and faster now means massively parallel). To compute the gravitational force and potential for a single interaction takes 28 floating point operations (here we count a division as 4 floating point operations and a square root as 4 floating point operations). A typical gravitational N-body simulation takes at least 1000 timesteps. Thus, when using the simple double loop over particles algorithm one requires $28 \times \frac{N^2}{2} \times 1000$ operations to complete a simulation. For 10 million particles this gives 1.4×10^{18} operations. A mythical teraflop machine (which by definition can compute 10^{12} floating point operations per second) would require 16 days to complete such a simulation. For current parallel machines that have a peak speed of around 50 Gflops, it would take nearly a year to complete the simulation. Thus, it should be clear that the simple brute-force approach is not a viable option for large N-body simulations (although using specialized hardware such as GRAPE [1] makes it a possibility).

The second approach is to try to develop better algorithms that can solve problems to the desired accuracy using much less computational power. In this paper we show that a moderately fast workstation running a treecode can update the forces on 10^5 particles in 50 seconds. The same 10 million particle simulation that would take a year on on a 1024 processor Thinking Machines CM-5 or Intel Paragon would take about 60 days on the workstation running a treecode. Ultimately, the most powerful method will be a combination of these two approaches—a sophisticated algorithm running on a parallel machine. However, a problem with the all-out approach is that the implementation of complicated numerical and computational methods on parallel computers is difficult. The sorry state of current operating systems and languages makes it almost impossible for scientists who do not have the time to become experts in parallel programming.

Two situations arise again and again in a variety of particle algorithms:

1. Finding neighbor lists for short-range interactions.

2. Computing global sums for long-range interactions.

For example, the problem of finding neighbors within the cutoff radius of a Lennard-Jones potential in a molecular dynamics simulation is qualitatively the same as finding neighbors in an SPH simulation. Similarly, the Biot-Savart summations that appear in vortex dynamics simulations are essentially the same as the Newtonian interactions that occur in astrophysics. One simply has a "vector mass" to contend with that adds somewhat to the complexity, but little to the essential underlying algorithm. Treecodes offer efficient and parallel solutions to both these situations which transcend the individual problem domains.

Software that is portable between different disciplines is an elusive but highly desirable commodity. It is virtually guaranteed that a project focused exclusively on a particular problem will not produce software that can easily be used outside that discipline. Appropriate abstractions do not emerge without careful analysis and design. We have specifically designed the software described here so that it can be used in a variety of areas. We use a single implementation of the underlying data structures to support all of these "applications." These tasks are diverse enough to require a careful design of interfaces and libraries in such a way that the "physics" is cleanly separated from the "data structures." We believe that the effort of designing a clean, highly modular implementation has not only saved us time (since we have been re-using our own software in separate sub-problems) but will also allow us to leverage our work to speed the development of high-quality parallel software in a variety of unrelated fields.

In this paper, we first describe in detail how an oct-tree based code may be applied to the neighbor finding problem in smoothed particle hydrodynamics. We then mention several other items that are relevant to implementation of an SPH code on a parallel machine. The next section desribes error bounds that are useful for multipole methods. We then turn to the description of the parallel implementation of the generic hashed oct-tree domain decomposition and tree building routines. The next sections present our strategy for making the code portable across a wide variety of computational platforms, and timings of the code for several problems on different parallel and sequential machines. The final sections briefly describe two other applications to which our code has been applied, a vortex particle method and a panel method.

2 Smoothed Particle Hydrodynamics

Smoothed particle hydrodynamics was introduced by [2, 3] and was combined with a hierarchical tree method by [4, 5]. The derivation of the SPH formalism has been presented elsewhere [6, 7, 8], and is beyond the scope of this paper. Our implementation of SPH follows that described in [7] for the most part. We jump immediately to the bottom line, which is a formula for the acceleration on particle i from the rest of the system:

$$\frac{d\vec{v}_i}{dt} = -\sum_{j=1}^{N} m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} \right) \vec{\nabla}_i W(|\vec{r}_i - \vec{r}_j|, h). \tag{1}$$

Here, m_j is the mass of particle j and h is a smoothing length that may or may not depend on the particular indices (i, j). The density at the position of each particle, ρ, is defined later in Eq. 5. The pressure, P, is derived from the equation of state for the material, i.e. $P = \frac{2}{3}U\rho$ for a perfect gas. The sum over j extends to all particles, but in practice a kernel with compact support is used, for which contributions from sufficiently distant particles vanish, i.e. the spline kernel,

$$W(r,h) = \frac{1}{\pi h^3} \begin{cases} 1 - \frac{3}{2}v^2 + \frac{3}{4}v^3 & \text{if } 0 \le v = r/h < 1 \\ \frac{1}{4}(2-v)^3 & \text{if } 1 \le v = r/h < 2 \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Thus, a fundamental requirement of any SPH algorithm is that of finding all particles j that contribute to the sum (the *neighbors* of particle i). The overall speed of an SPH simulation depends strongly on the ability to rapidly identify these particles.

2.1 Neighbor Finding

We define a particular neighbor finding problem as follows: associate with each particle i, a smoothing length h_i. For each particle i, find all other particles j that satisfy,

$$r_{ij}^2 < (h_i + h_j)^2. \tag{3}$$

This definition is of "mutual nearest neighbors" which is desirable to maintain pairwise symmetry in the interactions when using variable smoothing lengths.

The naive implementation of a neighbor finding routine uses a loop over the other $N-1$ particles in the system, comparing each distance to the cutoff radius of the kernel. This is done for each of the N particles in the system, which results in a number of operations that goes like $N(N-1) \sim N^2$ for large N. Since the hydrodynamics operations scale nearly like N, the computational time for N greater than a few thousand is completely dominated by the neighbor finding.

It has been said of SPH that it is almost as easy to write a 3-d code as it is to write a 1-d code. This is perhaps true, if one uses the naive neighbor finding routine. However, finding neighbors efficiently is tremendously easier in one dimension, where the list of spatial ordinates can be sorted. Then, all neighbors of any selected particle will form a contiguous group around the chosen particle, and may be identified with little effort. It should be clear that the time spent sorting the list is made up many times over by the gain in efficiency in locating the proper particles. In higher dimensions, the proper procedure is not as easy as a simple sort.

It is not difficult improve on the N^2 behavior of the search. Perhaps the simplest "smart" neighbor finding algorithm places particles into a regular lattice cells whose size is equal to twice the smoothing length (we assume here that all particles have equal or almost equal smoothing lengths). The neighbors of any given particle are then guaranteed to be in one of eight cells. On average, if a particle has n neighbors within a sphere of radius $2h$, then the eight cells that are searched will have $48n/\pi \approx 15n$ bodies that must be tested to find those n neighbors.

An alternative is to use an oct-tree (or quad-tree in 2-d). The oct-tree data structure collects particles into cubical regions called "cells". The cube that contains the entire system is called the "root." The root cell contains eight subcells that represent the sub-volumes obtained by dividing each of the dimensions of the root cell in half. The procedure is recursively continued on the subcells, until each cell is either empty or contains one particle. Tree construction is discussed further in section 5.

The basic idea in using a tree data structure for the neighbor finding problem is that one can eliminate large sets of particles with a single distance calculation. We borrow the terminology of the multipole treecodes, and refer to a particle "interacting" with a cell. In the case of neighbor-finding, an "interaction" is a determination of whether the contents of the cell can be eliminated from consideration as neighbors of the particle. In order to determine if all particles in the cell can be eliminated from consideration as neighbors, the distance between the center of the cell and the particle (r_{ij}) is compared with a distance made up of the smoothing length of particle i, the extent of the cell (b^{max}), and the "overlap," that accounts for the additional extent of the particles beyond b^{max} due to their smoothing lengths (see Fig. 1). Specifically, if

$$r_{ij}^2 > (h_i + b_j^{max} + max(h_j))^2 \tag{4}$$

then cell j can contain no neighbors of i. It might be more efficient to calculate the actual overlap in the tree construction stage directly (rather than using the upper bound of $max(h_j)$), since this would provide the tree traversal stage with better opportunities to prune cells from the tree. This idea has thus far not not been implemented.

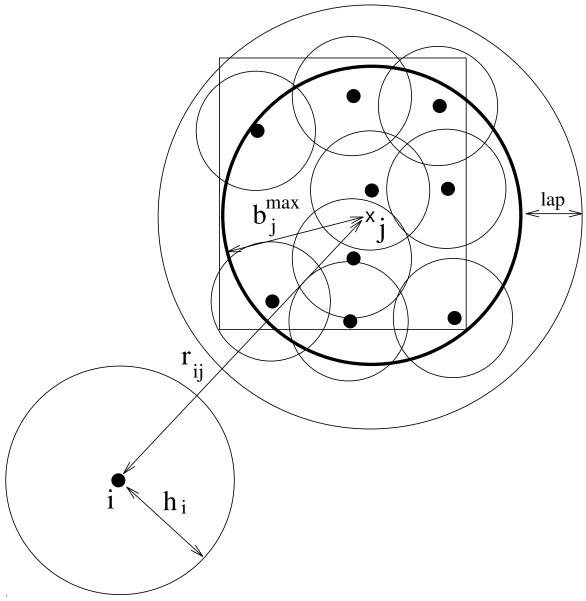

Figure 1: One can determine if a cell j (denoted by the square box) contains any neighbors of particle i by comparing the distance between the centroid of the cell and the particle (r_{ij}) with sum of the extent of the particles from the centroid (b_j^{max}), the maximal overlap of the smoothing lengths within j (the overlap), and the smoothing length of particle i (h_i). In this case, cell j contains no neighbors of particle i.

The efficiency of the method can be gauged by comparing the total number of distance interactions involved in identifying neighbors to the number of neighbors identified. Typically, the number of rejected particles and cells is a factor of 10 larger than the number accepted (i.e. each particle has about 60 neighbors, and 600 distance calculations are necessary to identify those 60. This is comparable to the simple grid-based algorithm described above, although the programming complexity and overhead of the grid-based algorithm is much lower. On the other hand, the grid based algorithm does not perform well on highly clustered systems or systems with highly variable smoothing lengths.

In any event, we may note that the number of floating point operations (flops) in each distance calculation is 11, while about 120 flops are required for each SPH calculation (i.e., for each neighbor). Thus, even an infinitely fast neighbor finding routine would only reduce the number of floating point operations vis-a-vis our current implementation by about a factor of two.

An adaptation that would likely reduce the tree traversal overhead a great deal would be to stop refining the tree when a cell has less than p particles. The cell would contain an array of the p particles. Then, all p particles could be tested at once in a well-vectorized routine in a manner identical to the search over a cell in a gridded calculation.

2.2 Updating the Forces

One computational strategy for updating the SPH forces is to initially update each particle's density (ρ) from,

$$\rho_i = \sum_{j=1}^{N} m_j W(|\vec{r}_i - \vec{r}_j|, h), \tag{5}$$

since ρ is required to update the pressures and densities that are used in the force equation (Eq. 1). One can save the neighbor lists from this traversal for re-use in the force calculation stage, or do an entirely new tree traversal. Saving the neighbor lists has the serious disadvantage of using a large amount of memory (assuming 50 neighbors per particle, with each particle labeled by an integer, the neighbor list memory overhead would amount to at least twice as much as the particle data itself). On a parallel machine, both possible strategies are somewhat problematic, since the ρ for off-processor data will change between the two sub-steps, necessitating a complete tree update, or some scheme for refreshing the ρ for particles from external processors.

However, one can avoid the ρ finding stage by using the continuity equation,

$$\frac{d\rho_i}{dt} = \sum_{j=1}^{N} m_j(\vec{v}_i - \vec{v}_j)\vec{\nabla}_i W(|\vec{r}_i - \vec{r}_j|, h) \tag{6}$$

to update ρ. This is in fact the preferred method for modeling solids with SPH, since the density at an edge is maintained at the proper value. The disadvantage is that mass will no longer be perfectly conserved due to numerical errors in the evolution of the continuity equation. The ideal solution for a parallel algorithm seems to be to use the continuity equation to produce an estimate of ρ for use in the force calculation, but use the sum over neighbors method during the force calculation stage to evaluate ρ exactly. This is the strategy we employ, except on the first timestep, where the sum over neighbors approach is used to initialize ρ.

One might think that by using symmetry in the particle forces, the number of SPH interactions could be reduced by a factor of two, thus increasing the computation speed considerably. One can, for example, update the force on both particle i and j when $i < j$ (this assumes the particles have been numbered somehow), but not when $i > j$. This scheme was implemented but no speed improvement was obtained. Our understanding of this phenomenon is that the additional writes to memory saturated the memory bandwidth and the CPU was not kept busy, i.e., the program became memory-bound. It is possible that with a detailed understanding of the memory performance characteristics of a particular processor, it would be possible to code the force routine so that the additional factor of two in speed would be realized.

2.3 Smoothed Particle Interpolation

A generalization of SPH called smoothed particle interpolation (SPI) has been made by Laguna [9], that is suitable for solving general hyperbolic and parabolic equations. The primary modification from conventional SPH is that in order to compute an n-th order derivative one requires a kernel of class C^n. Thus, the smoothed approximation to the Laplacian requires a C^2 kernel. The popular spline kernel (Eq. 2) is not suitable because it has a discontinuous second derivative. For this reason, the Gaussian kernel

$$W(r,h) = \frac{1}{\pi^{3/2}h^3}\exp(-(r/h)^2) \tag{7}$$

is used. One evident disadvantage of this kernel is that it does not have compact support. Thus, a Gaussian kernel requires a larger number of neighbors to achieve comparable accuracy to the spline kernel.

This disadvantage of a Gaussian kernel may be avoided by using the same approach as that of a treecode computing long-range forces (i.e. gravity) by using multipole expansions. A particle would then interact with other particles that are nearby, and groups of particles that are more distant. It allows the possibility of groups interacting with groups (i.e. an $O(N)$ method, but now with long-range forces). It is interesting to note that with a typical force accuracy criterion of 1% on a relatively uniform distribution, it is possible to calculate the gravitational forces using a treecode about three times faster than the SPH forces. This is partly due to the fact that the gravitational code can interact with multi-particle cells at a distance roughly equivalent to the smoothing radius, while an SPH code with a compact-support kernel must do a lot of work to separate particles just inside the smoothing radius from those just outside. This suggests that an SPH code using an exponential or Gaussian kernel, and performing multipole expansions as described in the next section could perhaps perform better (for similar accuracy) than a "short-range" spline kernel.

3 Multipole Methods

In this section, we review some of the mathematics behind multipole methods, and present error bounds suitable for computing local expansions arising from distributed sources for arbitrary Green's functions. In our earlier work [10, 11], we restricted our attention to methods which computed only Body-Cell or Body-Body interactions (where the sources are single particles, i.e. not distributed). Here we extend the analysis to include Cell-Cell interactions.

Treecodes have been used extensively to approximate gravitational and electrostatic interactions. These Newtonian or Coulomb interactions are a special case of a more general formulation that can be applied to any pairwise interaction or Green's function. Even short-range interactions, i.e., those which can be adequately approximated as vanishing outside some radius, can be efficiently and seamlessly handled with this formalism.

Consider a configuration of sources as in Fig. 2. The sources are contained within a "source" cell, $\mathcal{V}$ of radius b_{max}, while the field is evaluated at separation $\boldsymbol{\Delta}$ from $\boldsymbol{x}_0$, the center of "sink" cell $\mathcal{W}$.

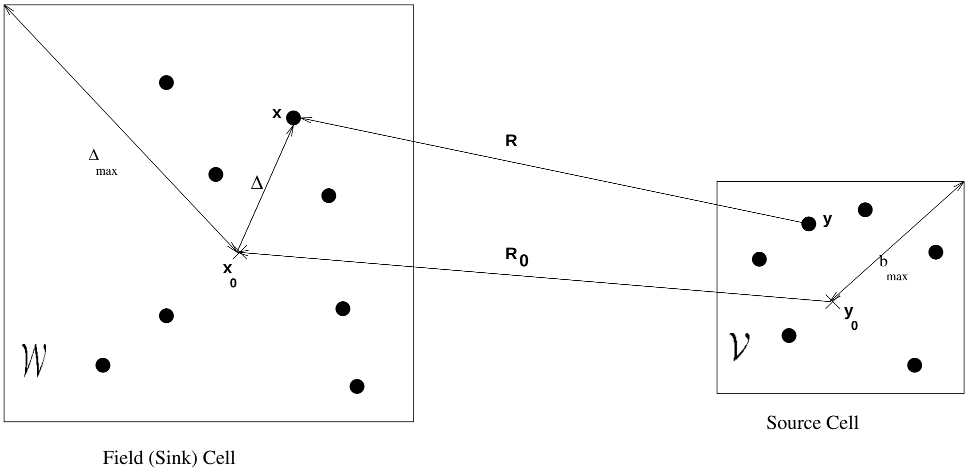

Figure 2: An illustration of the relevant distances used in the error bound equation.

In terms of an arbitrary Green's function, G, the field is:

$$\phi(\boldsymbol{x}) = \int_{\mathcal{V}} d\boldsymbol{y} G(\boldsymbol{x}-\boldsymbol{y})\rho(\boldsymbol{y}) \tag{8}$$

$$= \int_{\mathcal{V}} d\boldsymbol{y} G((\boldsymbol{x}_0-\boldsymbol{y}_0)-(\boldsymbol{y}-(\boldsymbol{y}_0+\boldsymbol{\Delta})))\rho(\boldsymbol{y}) \tag{9}$$

Expanding G in a Taylor series around $\boldsymbol{R}_0 = \boldsymbol{x}_0 - \boldsymbol{y}_0$ leads to the multipole expansion:

$$\phi(\boldsymbol{x}) = \sum_{n=0}^{p} \frac{(-1)^n}{n!} \partial_{(n)} G(\boldsymbol{R}_0) \odot \boldsymbol{M}^{(n)}(\boldsymbol{y}_0 + \boldsymbol{\Delta}) + \Phi_{(p)}(\boldsymbol{x}) \tag{10}$$

where $\Phi_{(p)}$ is the error term, and the moment tensor $\boldsymbol{M}^{(n)}(\boldsymbol{z})$ is defined relative to a center, $\boldsymbol{z}$ as:

$$M^{(n)}(\boldsymbol{z}) = \int d\boldsymbol{y}(\boldsymbol{y}-\boldsymbol{z})^{(n)}\rho(\boldsymbol{y}) \tag{11}$$

We have used a notational shorthand in which $\boldsymbol{v}^{(n)}$ indicates the n-fold outer product of the vector $\boldsymbol{v}$ with itself, while $\odot$ indicates a tensor inner-product and $\partial_{(n)}G$ indicates the rank-n tensor whose components are the partial derivatives of G in the cartesian directions.

We can further expand the result by writing $\boldsymbol{M}^{(n)}(\boldsymbol{y}_0+\boldsymbol{\Delta})$ as a sum over powers of the components of $\boldsymbol{\Delta}$, and then recollecting terms to obtain:

$$\boldsymbol{F}^{m,p} = \sum_{n=m}^{p} \frac{(-1)^{n-m}}{(n-m)!}\partial_{(n)}G(\boldsymbol{R}_0)\odot \boldsymbol{M}^{(n-m)}(\boldsymbol{y}_0) \tag{12}$$

$$\phi(\boldsymbol{x}) = \sum_{m=0}^{p}\frac{1}{m!}\boldsymbol{\Delta}^{(m)}\odot \boldsymbol{F}^{m,p} + \Phi_{(p)}(\boldsymbol{x}) \tag{13}$$

$$\nabla\phi(\boldsymbol{x}) = \sum_{m=0}^{p}\frac{1}{m!}\boldsymbol{\Delta}^{(m)}\odot \boldsymbol{F}^{m+1,p+1} + \nabla\Phi_{(p)}(\boldsymbol{x}) \tag{14}$$

The expression for $\nabla\phi$ follows from taking the derivative of both sides Eq. 8 and following the same steps that lead to Eq. 13.

The summation over n represents the "interaction" between a source cell, $\mathcal{V}$, represented by the multipole moment tensors $\boldsymbol{M}^{(n-m)}$, and a sink cell, $\mathcal{W}$, represented by the field tensors $\boldsymbol{F}^{m,p}$. The summation over m represents the "evaluation" of a Taylor series at the evaluation point $(\boldsymbol{x}_0+\boldsymbol{\Delta})$.

Using the Lagrange form of the remainder in a Taylor series, it is straightforward to derive the following error bound:

$$\left|\Phi_{(p)}(\boldsymbol{x})\right| \leq \frac{1}{(p+1)!}\max_{\boldsymbol{R}\in\mathcal{W}-\mathcal{V}}\left|\left|\partial_{(p+1)}G(\boldsymbol{R})\right|\right| B_{(p+1)}(\boldsymbol{y}_0+\boldsymbol{\Delta}) \tag{15}$$

where the max is taken over all vectors $\boldsymbol{R}=\boldsymbol{x}-\boldsymbol{y}$ with $x\in\mathcal{W}$ and $y\in\mathcal{V}$,

$$B_{(p+1)}(\boldsymbol{z}) = \int_{\mathcal{V}} d\boldsymbol{y}\,|\boldsymbol{y}-\boldsymbol{z}|^{p+1}\rho(\boldsymbol{y}). \tag{16}$$

and the norm of a rank-n tensor $\boldsymbol{T}$ is defined as:

$$||\boldsymbol{T}|| = \max_{|v|=1}\left|\boldsymbol{T}\odot\boldsymbol{v}^{(n)}\right|. \tag{17}$$

Furthermore,

$$B_{(p+1)}(\boldsymbol{y}_0+\boldsymbol{\Delta}) \leq \sum_{m=0}^{p+1}\binom{p+1}{m}\Delta_{max}^{m}B_{(p+1-m)}(\boldsymbol{y}_0) = \mathcal{B}_{(p+1)}(\Delta_{max}) \tag{18}$$

Eqs. 15 and 18 give a bound on the error in terms of properties of the source cell ($B_{(i)}(\boldsymbol{y}_0)$), the sink cell ($\Delta_{max}$), and the separation between them ($\left|\left|\partial_{(p+1)}G(\boldsymbol{R})\right|\right|$), so it can be used to decide, dynamically, when the multipole approximation is accurate enough to be used between a pair of cells.

The error bound is essentially a precise statement of several intuitive ideas. Interactions are more accurate when:

- The interaction is weak, or it is well-approximated by its lower derivatives. (small $\left|\left|\partial_{(p+1)}G\right|\right|$).
- The sources are distributed over a small region (smaller b_{max}).
- The field is evaluated near the center of the local expansion (smaller Δ_{max}).
- More terms in the multipole expansion are used (larger p).
- The truncated multipole moments are small (smaller $B_{(p+1)}$).

3.1 Special case, $p = 1$

The case $p = 1$ is particularly important. It corresponds to the dipole approximation in electrostatics, and the monopole approximation around the center-of-mass in gravitation. It is easy to program, as it involves only vectors and scalars, and hence avoids issues like how to efficiently store and manipulate high-order symmetric tensors. The summations in Eq. 13 and 12 can be "unrolled", and the result is quite simple:

$$F^{0,1} = M^{(0)}G(\boldsymbol{R}_0) - \nabla G(\boldsymbol{R}_0)\cdot\boldsymbol{M}^{(1)} \tag{19}$$

$$\boldsymbol{F}^{1,1} = \nabla G(\boldsymbol{R}_0)M^{(0)} \tag{20}$$

$$\phi(\boldsymbol{x}) = F^{0,1} + \boldsymbol{\Delta}\cdot\boldsymbol{F}^{1,1} \tag{21}$$

In these formulae, $M^{(0)}$ is the total "mass" or "charge" in the region $\mathcal{V}$, while $\boldsymbol{M}^{(1)}$ is the dipole moment. If it is possible to arrange that $\boldsymbol{y}_0$ is the center-of-mass of $\mathcal{V}$, then the dipole moment vanishes by construction, and the term containing $\boldsymbol{M}^{(1)}$ can be neglected.

If the Green's function is spherically symmetric, i.e., $G(\boldsymbol{R}) = G(R)$, then the results simplify even further, with $\hat{\mathbf{e}}_{\boldsymbol{R}_0}$ a unit-vector in the direction of $\boldsymbol{R}_0$, we have:

$$F^{0,1} = M^{(0)}G(R_0) - G'(R_0)\hat{\mathbf{e}}_{\boldsymbol{R}_0}\cdot\boldsymbol{M}^{(1)} \tag{22}$$

$$\boldsymbol{F}^{1,1} = G'(R_0)\hat{\mathbf{e}}_{\boldsymbol{R}_0}M^{(0)} \tag{23}$$

$$\phi(\boldsymbol{x}) = F^{0,1} + \boldsymbol{\Delta}\cdot\boldsymbol{F}^{1,1} \tag{24}$$

with the following error bounds:

$$||\partial_i\partial_j G(\boldsymbol{R})|| = \max\left(\left|\frac{G'(R)}{R}\right|, |G''(R)|\right) \tag{25}$$

$$|\Phi(\boldsymbol{x})| \leq \frac{1}{2}\max_{\boldsymbol{R}\in\mathcal{W}-\mathcal{V}} ||\partial_i\partial_j G(\boldsymbol{R})||\, B_{(2)}(\Delta_{max}) \tag{26}$$

3.2 Newtonian potential, Gravity

If we further specify that the Green's function is the Newtonian potential, $G(\boldsymbol{R}) = 1/R$, the results for $p = 1$ are:

$$F^{0,1} = \frac{1}{R_0}\left(M^{(0)} + \frac{1}{R_0}\hat{\mathbf{e}}_{\boldsymbol{R}_0}\cdot\boldsymbol{M}^{(1)}\right) \tag{27}$$

$$\boldsymbol{F}^{1,1} = -\frac{1}{R_0^2}\hat{\mathbf{e}}_{\boldsymbol{R}_0}M^{(0)} \tag{28}$$

$$\phi(\boldsymbol{x}) = F^{0,1} + \boldsymbol{\Delta}\cdot\boldsymbol{F}^{1,1} \tag{29}$$

$$|\Phi(\boldsymbol{x})| \leq \frac{1}{R_0 - (b_{max} + \Delta_{max})} \frac{\mathcal{B}_{(2)}(\Delta_{max})}{R_0^2} \tag{30}$$

$$|\nabla\Phi(\boldsymbol{x})| \leq \frac{1}{(R_0 - (b_{max} + \Delta_{max}))^2} \left(\frac{3\mathcal{B}_{(2)}(\Delta_{max})}{R_0^2} - \frac{2B_{(3)}}{R_0^3} \right) \tag{31}$$

The bound on the error is tighter than that implied by Eq. 15. It follows from the full integral form of the Taylor series remainder.

Using the error bounds derived above, one can devise a variety of efficient criteria for determining which cells a particle should interact with, subject to a user-specified error bound. Given an upper limit on the acceptable error (relative, absolute, summed, in the potential, in the gradient, etc.), one can compute the maximum error that could be introduced by using the approximation and thereby determine whether the approximation is acceptable. The information required to make this determination takes the form of easily computed moments of the mass or charge distribution (strength) within the cell, i.e., the $B_{(p)}$. Computing this information takes place in the tree construction stage, and takes very little time compared with the later phases of the algorithm.

Our implementation of the $N \log N$ [12, 11] method associates a critical radius (r_c) with each cell, and demands that any particle that interacts with the cell lie outside the sphere defined by r_c. In this method, all sinks are individual bodies, so $\Delta_{max} = 0$. One may solve for r_c in the equation,

$$|\nabla\Phi(\boldsymbol{x}_0)| \leq err_{max}, \qquad \text{for} R_0 \geq r_c$$

where err_{max} is a user-specified absolute error tolerance. For the case of $p = 1$, the critical radius can computed analytically using Eq. 31 and the fact that $B_3 \geq 0$:

$$r_c \leq \frac{b_{max}}{2} + \sqrt{\frac{b_{max}^2}{4} + \sqrt{\frac{3B_2}{err_{max}}}}.$$

B_2 is simply the trace of the quadrupole moment tensor. In more general cases (using a better bound on B_3, or with $p > 1$), r_c can be computed from the error bound equation (Eq. 31) using Newton's method. The overall computational expense of calculating r_c is small, since it need only be calculated once for each cell. Furthermore, Newton's method need not be iterated to high accuracy. The multipole acceptance criterion (MAC) then becomes

$$R_0^2 > r_c^2 \tag{32}$$

for each displacement R_0 and critical radius r_c (Fig. 3). This is computationally very similar to the Barnes-Hut opening criterion [13], where instead of using a multiple of the box size, s/θ, we use the distance r_c, derived from the contents of the cell and the error tolerance.

We have also implemented a gravitational code that computes local expansions for non-terminal sinks, i.e., $\Delta_{max} > 0$. Here, we use the r_c test as a first-cut, eliminating most of the interactions with a very fast test. If the separation between cells exceeds r_c, then we evaluate the right-hand-side of Eq. 31 and compare the result with err_{max} to determine if the multipole approximation is acceptable. Only if it is acceptable is the local expansion computed and stored in the sink cell. If the approximation is unacceptable, we split the larger of the sink and source cells, i.e., we split the sink iff $\Delta_{max} > b_{max}$.

4 Domain Decomposition

When computing in parallel, there are two computational tasks that must be done well in order to succeed. The first is splitting up the data among the processors. The second is efficiently accessing data that is in

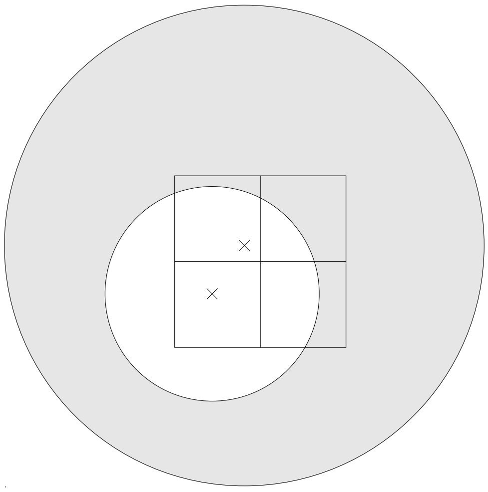

Figure 3: The critical radii of a cell and one of its daughters are shown here as circles. For the specified accuracy, a particle must lie outside the critical radius of a cell. The shaded region shows the spatial domain of all particles that would interact with the lower left daughter cell. Those particles outside the shaded region would interact with the parent cell, and those within the unshaded region would interact with even smaller cells inside the daughter cell.

the memory of another processor. The use of data-parallel languages attempts to make the first task easy, and hide the second task from the programmer. In cases where the data layout is fairly simple, and the communications patterns are regular this is an excellent approach. However, when dealing with dynamic, irregular data structures, data parallel programming can be difficult, and the end result may be a program with sub-optimal performance. In the case of parallel treecodes, we decided that having as much control over the distribution and communication as possible would result in the most efficient program. This means using a message-passing approach, where all communication takes place in explicit function calls. The tradeoff for better performance is a more complicated algorithm. A conscious attempt was made to minimize the programming effort involved by developing a simple unified model of how to domain decompose and remotely access particle-type data.

For many physical problems, the interactions are primarily local (and in the case of tree methods, non-local interactions are approximated in a manner that ends up making most interactions local). This means that the amount of data that must be communicated to a spatial "domain" of particles will be proportional to the surface area of that domain. The optimal domain decomposition will split the work involved into equal sized pieces, while minimizing the surface area of each processor domain. Several varieties of domain decompositions have been used. In our first generation code, we used orthogonal recursive bisection [14]. This splits the work in half along the first spatial dimension of the entire domain, and then independently splits the two halves along the second dimension and so on until each processor has a piece of the problem. The disadvantage to this method for treecodes comes when one wishes to build a global tree representation of the system or retrieve data from another processor, since the domain decomposition does not directly correspond to the tree structure.

We use a method based on Morton ordering (see Fig. 5) that converts a d-dimensional set of data points into a 1-dimensional list, while maintaining as much spatial locality in the list as possible. This allows us to neatly domain decompose any set of particle data. The idea is simply to cut the one-dimensional list of sorted body key ordinates (keys are discussed in the next section) into N_p (number of processors) equal pieces, weighted by the amount of work corresponding to each body. The work for each body is readily

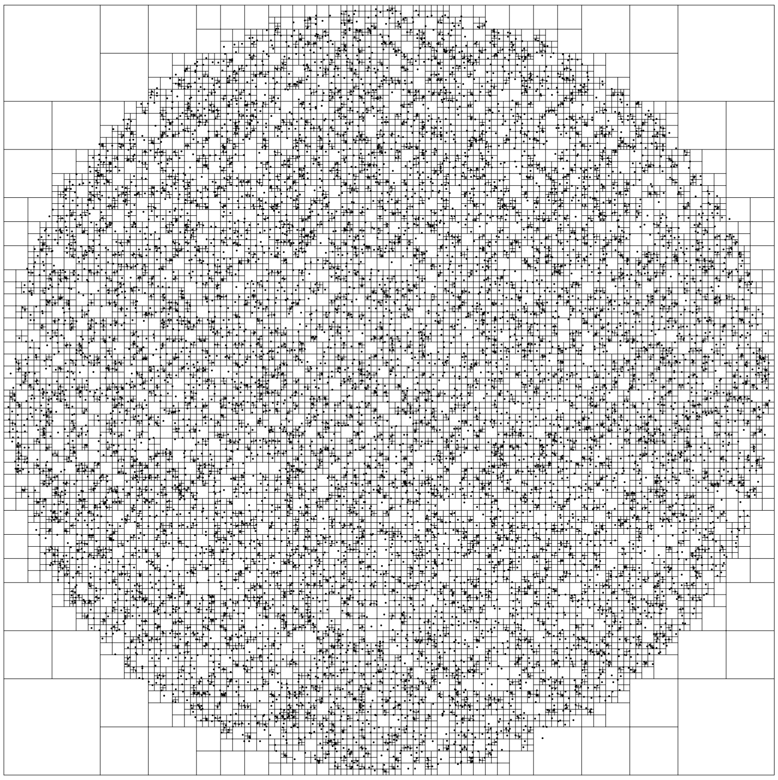

Figure 4: A representation of a regular tree structure in two dimensions (a quad-tree) that contains 10 thousand particles that are randomly distributed.

approximated by counting the number of interactions the body was involved in on the previous timestep.

The implementation of the domain decomposition is practically identical to a parallel sorting algorithm, with the modification that the amount of data that ends up in each processor is weighted by the work associated with each item. We have also begun to investigate using the tree structure itself to provide pertinent information for the next domain decomposition stage (an idea that was first proposed in [15]), since in an $O(N)$ method the work is not so much associated with particles as it is with cells.

5 Tree Construction

In a conventional oct-tree data structure, the topology of the tree is indicated by storing in each parent cell a set of links that point to the daughter cells. This allows a tree traversal to begin from the root of the tree and proceed downward (here we are using the computer science convention in which trees are upside down; the root of the tree is at the top and the and the more refined parts are at the bottom). The traversal does not, however, allow one to proceed from the bottom upward, or across to a spatial neighbor without adding additional pointers to the data structure. It also does not support random access to a particular cell in the tree. In addition, on a distributed-memory parallel machine, some special handling must occur when the daughter cell (which would otherwise be pointed to) is in another processor. These issues make implementation of a pointer based oct-tree on a parallel machine unwieldy.

We have invented a scheme to label each possible cell in a distributed oct-tree data structure with a multi-bit *key*. This labeling scheme implicitly defines the topology of the tree, and makes it possible to compute easily the key of a parent, daughter, or boundary cell for a given key. In order to translate the key

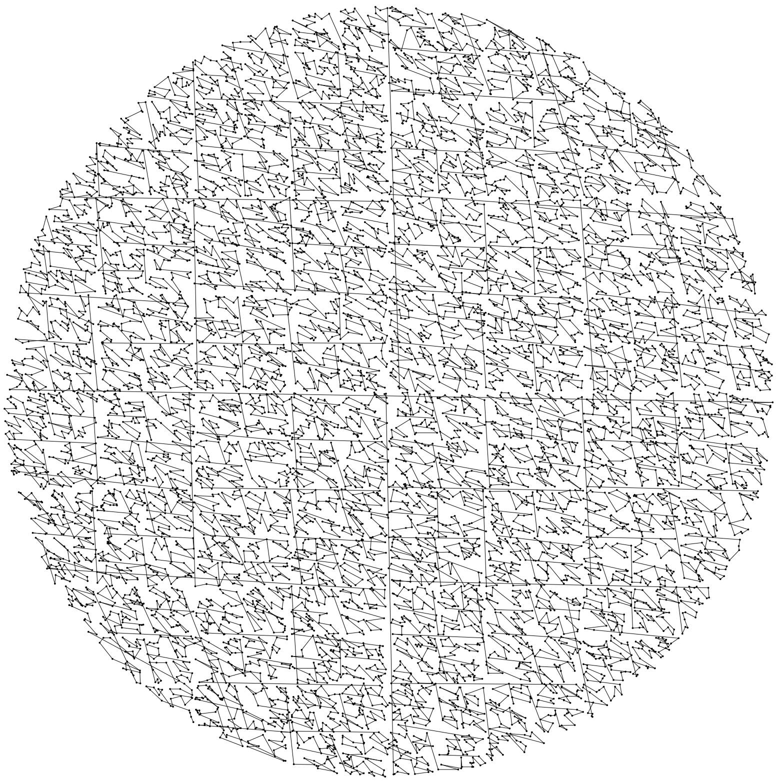

Figure 5: The path indicates the one-dimensional self-similar path that is induced by the map of interleaved bits (Morton order). The domain decomposition is achieved by cutting the one-dimensional list into N_p pieces.

into a pointer to the location where the cell data is stored, we use a hash table. This level of indirection through a hash table can also be used to catch accesses to non-local data, and allows us to request and receive data from other processors. This is the basis of the hashed oct-tree method (see [16] for further details).

It is very important for many parallel computational algorithms to efficiently represent irregular data structures. While it is possible to represent cleanly and efficiently regular structured grids on a parallel machine, a mechanism to handle irregular data structures is not so easily developed. When thinking of a generic structure for use in an adaptive finite difference code, for example, it is essential to be able to identify the grid points that border a particular grid cell. We designed the hashed oct-tree data structure for use with an N-body treecode, but also with the hope that it would be useful for other parallel algorithms, such as a spatially adaptive finite difference code. Much of the programming complexity of these types of algorithms is due to the identification of grid cells that are on the borders of coarse and fine sub-grids. If one restricts the basic data structure to an oct-tree, most of the benefit of spatial adaptivity may be obtained, and the complexity of neighboring grid cell finding is largely removed.

Our keys are constructed from the body co-ordinates in d-dimensional space by converting floating point numbers in the domain $(\vec{r}_{min}, \vec{r}_{max})$ to integers, where the $\vec{r}$'s define the opposite corners of the cube that bounds the particle distribution. We map the $(l_k - 1)/d$ most significant bits of the integers to a key that is l_k bits long by interleaving the bits one at a time (Fig. 6). l_k is stored as a vector of integers and may vary depending on requirements, but we typically use $l_k = 64$ bits. Note that we place no restriction on the dimension of the space, although we are physically motivated to pay particular attention to the case of $d = 3$. In this case, we use 21 bits from each integerized co-ordinate and the key fits into a single 64 bit integer or a pair of 32 bit integers. This results in an oct-tree with 21 levels, which we have found to be sufficient to represent highly clustered data sets with over ten million particles.

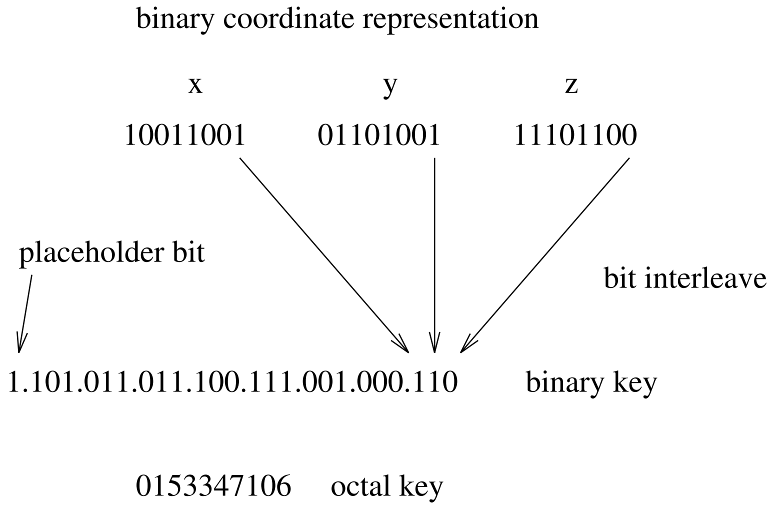

Figure 6: An illustration of the key mapping. Bits of the coordinates are interleaved and a place-holder bit is prepended to the most significant bit. In this example, the 8-bit x, y and z values are mapped to a 25-bit key. In practice, we use 21 bit values and a 64-bit key

Apart from the choice of origin and coordinate system, this is identical to Morton ordering (also called Z or N ordering, see Chapter 1 of [17] and references therein, and also [12]). By a suitable modification of this scheme, we can represent internal nodes of an oct-tree using the same type of key. We prepend an additional 1-bit to the most significant bit of every key (the place-holder bit), and can then represent all higher level nodes in the tree in the same key space. Without the place-holder bit, there would be an ambiguity among keys where the most significant bits are all zeroes. A two-dimensional representation of such a tree is shown in Fig. 7. The key space is very convenient for tree traversals. In order to find daughter nodes, the parent key is left-shifted by d bits, and the result is added (or equivalently `OR`'ed) to daughter numbers from 0 to $2^d - 1$. Also, the key retrieval mechanism is much more flexible in terms of the kinds of accesses that are allowed.

5.1 Hashing

Each key identifies the location in the tree of a particular collection of data (for instance, the mass and position of a particle or cell). However, the location in memory of the data is not defined. To retrieve the data corresponding to a particular key, the key must be translated to a pointer that represents the memory location containing the data. The number of possible keys is very large, and to store them in a list for a 21-level oct-tree would require $2^{63} \approx 10^{19}$ storage locations. This would of course be extremely wasteful, since the number of keys actually needed for a simulation is within a factor of two or so of the number of particles in the simulation. A standard method for this type of indexing problem is to find a way to map values from the very large set of possible keys into a much smaller index set which is used as an index into a *hash table*. It is desirable for this mapping (called hashing) to spread the keys uniformly throughout the smaller set. We use a very simple hashing function, which is to `AND` the key with the bit-mask $2^h - 1$, which selects the least significant h bits of the k bit key. When two or more keys map to the same location (a collision) the address is resolved by following a linked list of those keys with identical hash table addresses. If the desired key is found in the hash table, this access scheme is almost as fast as using a conventional

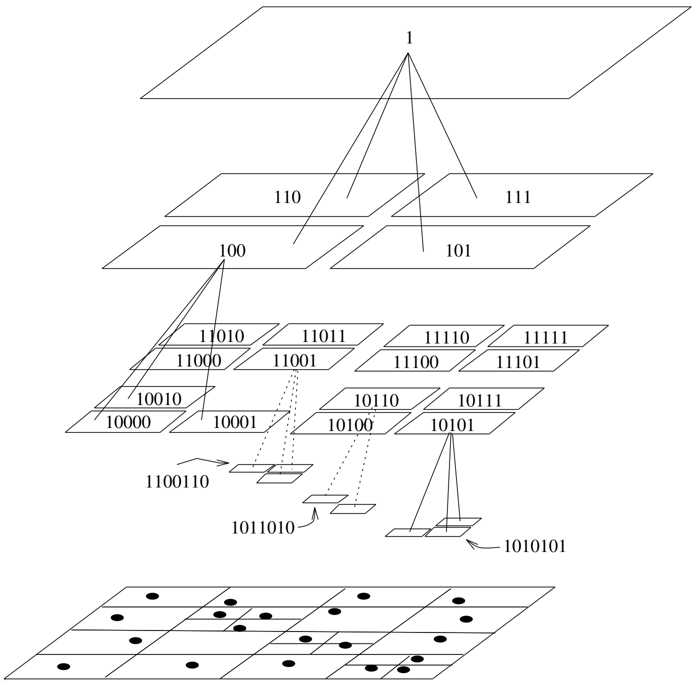

Figure 7: A quad-tree shown along with the binary key coordinates of the nodes. At the bottom is a "flat" representation of the tree topology induced by the 20 particles. The rest of the figure demonstrates the relation of the key coordinates at each level to the tree topology. Many of the links from parent to daughter cells are omitted for clarity.

pointer oct-tree. However, if several keys have mapped to the same location, retrieval performance will degrade in proportion to the number of keys that have collided. It is therefore quite important that the incidence of collisions be reduced to a minimum.

Our hashing scheme uses the simplest possible function; a one instruction bitwise `AND`. However, it is really the map of floating point coordinates into the key that performs what one usually would consider "hashing." The structure of the hierarchical key space and selection of the least significant bits of the key performs extraordinarily well in reducing the incidence of collisions. For the set of all keys that contain fewer significant bits than the hash mask, the hashing function will never result in a collision. This set of keys represents the upper levels of the tree, which tend to be accessed the most often. At lower levels of the tree (where the number of bits in a key exceeds the length of the hash mask), distinct keys can result in the same hash address. However, the properties of the map means these keys are spatially well-separated, and would not tend to be needed at similar stages of the tree traversal. Also, on a parallel machine many of the keys that would results in collisions become distributed to different processors (each of which has its own hash table).

5.2 Tree Construction in Parallel

After the domain decomposition, each processor has a disjoint set of bodies. The initial stage in parallel tree building is the construction of a tree made of the local bodies. A special case occurs at each processor boundary in the one-dimensional sorted key list, where the bodies at the edges of the list in adjacent processors could lie in the same cell. This is taken care of by sending a copy of each boundary body to the adjacent processor, that allows the construction of the proper tree nodes. Then, copies of *branch* nodes from each processor are shared among all processors. The branch nodes are defined as the smallest subset of cells that represent the entire domain of a processor. This stage is made considerably easier and faster since the domain decomposition is intimately related to the tree topology (unlike the orthogonal recursive bisection method used in our previous code [14]). The branches make up a complete set of cells that represent the entire processor domain at the coarsest level possible. These branch cells are then globally communicated among the processors. All processors can then "fill in" the missing top of the tree down to the branch cells. The address of the processor that owns each branch cell is passed to the destination processor, so the cell created is marked with its origin. A traversal routine can then immediately determine which processor to request data from when it needs access to the daughters of a branch cell. The daughters received from other processors are also marked in the same fashion.

6 Tree Traversal

The basic idea in all treecodes is to use some criterion to reject or approximate the effects of large numbers of bodies at a time. This may be a geometrical criterion, e.g. Fig. 1, or a complicated approximation, e.g., those in [11]. We have coded a generic tree traversal function which is dynamically configurable as a driver for either $O(N^2)$, $O(N \log N)$ or $O(N)$ traversals. The basic idea is to distinguish between a "source" tree and a "sink" (i.e., whatever the source is interacting with) tree. The source tree contains the bodies and their multipole moments that create the gravitational or electrostatic field in multipole codes. It is the collection of bodies from which neighbors must be found in SPH. A cell in the sink is a collection of bodies whose interactions are computed together. In the language of [18], a sink is the center of a "local expansion". Barnes' [19] simultaneous application of the multipole acceptability criterion to collections of bodies is similar in spirit to our use of sinks. Usually, when called from an application code, the `source` and `sink` arguments to `Traverse` will be identical, i.e., they will both be the root of the tree of bodies. However, they may in fact be different. For example, if one were interested in the gravitational force on only a subset of bodies, or at one or more locations at which there were no bodies, then one could construct a sink-tree consisting of only those locations at which the force was desired. The dual-traversal is:

```
Traverse(sourcelist, sink, Inherit, Interact){
  newsourcelist = sourcelist;
  while( newsourcelist is not empty ){
      sourcelist = newsourcelist;
      Interact(sourcelist, sink, resultlist);
      clear newsourcelist;
      for(each (result,source) pair){
        switch(result){
        case SPLIT_SRC:
           append children of source to newsourcelist;
           break;
        case SPLIT_SINK:
           append source to inheritlist;
           break;
        case FINISHED:
           break;
        }
      }
  }
  for( each child of sink ){
    Inherit(sink, child);
    Traverse(inheritlist, child, Inherit, Interact);
  }
}
```

The behavior is modified by changing the `Inherit` and `Interact` functions, that are passed as pointers to the generic tree-traversal routine. These functions encode all of the "physics" of the application, and are completely independent of the complexities of parallelism, tree management, and decomposition. Notice that `Interact` takes a list of sources as an argument, so it can, in principle, be vectorized.

In fact, the actual implementation is slightly different from the pseudo-code above because it is not necessary to store the "Inherited" information after a branch of the sink tree has been completed. Thus, the inherited information (e.g., local expansions) is stored in an auxiliary stack that grows and shrinks with the depth of the traversal of the sink-tree.

Furthermore, the pseudo-code hides much of the complexity that arises due to parallelism. On a distributed memory parallel machine, the pseudocode `append children to newsourcelist` may entail interprocessor communication to obtain the data associated with those children. The code handles this by sending a message requesting the necessary data and continuing with other branches of the traversal that do not depend on the requested data. The strategy is similar to what one would use if fast virtual shared memory and multi-threaded control were available. In the interests of portability and performance, the implementation is completely explicit, i.e., every "context switch" and every "virtual memory reference" appear explicitly in the code as branches and communication calls. They are not handled by operating system abstractions.

The `Interact` function has two missions: to compute "interactions" between sources and sinks, or, if that is not possible due to some "physics" constraint (e.g., accuracy, geometry, etc.) to determine whether the sink or source should be split to satisfy the appropriate constraints. Three interesting cases emerge from different choices in the `Interact` behavior:

1. $O(N^2)$, all-pairs behavior occurs if `Interact` always splits non-terminal sources and sinks. Of course, an all-pairs calculation is more efficiently done without using a tree at all.

2. If `Interact` splits all non-terminal sinks but allows interactions between terminal sinks (i.e., individual bodies) and non-terminal sources (i.e., multipoles), the result is similar to the $O(N \log N)$ behavior of [12, 11]. It is also possible to obtain the performance increase described in [19] by a slight modification to the opening criterion.

3. If `Interact` allows for interactions between non-terminal sinks (i.e., local expansions), and non-terminal sources (i.e., multipoles), the behavior is reminiscent of the Fast Multipole Method [18]. Notice, however, that the structure is much more general since cells of different sizes can, in principle, interact and accuracy can be enhanced either through high-order expansions or splitting of sources or sinks.

6.1 Memory Hierarchy and Access Patterns

Another factor to keep in mind is that the limiting path in performance is often the speed at which data can be accessed from main memory. The memory hierarchy in modern microprocessors is not forgiving of programs that access data at random throughout many megabytes of memory. The quasi-random access to widely separated memory locations during the tree traversal receives little benefit from a small on-chip cache, and can in fact overwhelm the translation look-a-side buffer (TLB) on microprocessors similar to the `i860`. This results in very poor performance from algorithms that have been designed without consideration of memory bandwidth and the limitations inherent in memory hierarchies that are intended to function with main memory significantly slower than the processor cycle time.

By making the access pattern more "local" and orderly, one can speed up a program significantly. A useful property of tree algorithms is that particles that are spatially near each other tend to have very similar cell interaction lists. By updating the particles in an order that takes advantage of their spatial proximity, we can reduce the number of memory accesses that miss the cache and TLB. A convenient and efficient ordering once again uses the same sorted key list used in the tree construction. By updating particles in the order defined by the key map (Fig. 5), we minimize the number of cache and TLB misses.

An additional technique to improve memory access speed is through the rearrangement of data in the linked list of collisions in the hash table. By moving data that has been recently accessed to the top of the linked list, it is possible to create a "virtual cache" by keeping often used data in the contiguous memory locations making up the hash table. This also allows one to obtain good performance with a hash table much smaller than one would naively expect.

The more extended memory hierarchy in a distributed memory parallel computer (possibly with virtual memory on each node) can benefit from this scheme as well. We wish to keep things for as long as possible in the fastest level of the hierarchy that includes registers, cache, local memory, other processors memory, and virtual memory. We could extend the "virtual cache" model even further, by erasing data that has come from another processor that has not been used recently. Although this has not been implemented, we expect that it will allow us to run significantly larger simulations, since the majority of the memory used now consists of copies of cells from other processors.

7 Portability

There are a variety of parallel computers that support the message-passing programming paradigm. A partial list of machines of this type includes the CM-5, Intel Paragon, nCUBE II, Cray T-3D and IBM SP-1. When developing a program to run on this type of machine, it makes little sense to use vendor-specific functions that restrict one to particular vendor or architecture. There have been attempts to establish de-facto standards with PVM and MPI, but we have found neither are sufficiently mature or availble in optimized form to meet our needs. It is a relatively simple matter to select a basic set of message-passing functions, and write a single system dependent file that implements each of these functions in the native operating system of a particular machine. One can add even further levels of indirection: we have implemented our message passing interface on top of PVM, for example. By keeping the communication library simple, we are able to implement it immediately on new machines, often months before other systems are in place.

As a further extension, we have implemented our message passing functions using UNIX User Datagram Protocol (UDP) sockets, which allows us to compute on any network of workstations supporting that communication protocol. The most important benefit of being able to run parallel programs on a network of workstations (or as multiple processes on a single machine) has been that we are able to use state-of-the-art debugging tools during program development.

8 Performance

In Table 1, we show benchmark results for several representative machines and problem sizes. The first benchmark problem is N particles of mass $1/N$ distributed randomly in a sphere of radius 1. The error bound for each partial interaction is set to 1% of M/R^2, which is .01 in this case. To confirm the accuracy of the method, the potential and force were calculated for a dataset with 100,000 particles using an exact N^2 algorithm, and these values were used to calculate maximum and rms errors. These errors are shown in Table 2. The second benchmark problem uses a highly clustered set of particles (an isothermal density distribution, $\rho(r) \sim 1/r^2$) and a fractional error bound was used, where the error for each partial interaction was limited to 1% of the total force on the particle. The total force was estimated from the force on the previous timestep, except for the first timestep, where it was estimated by using a lenient absolute error bound. The third benchmark problem is SPH+gravity for a random spherical distribution with a smoothing length that results in a mean number of neighbors of 65. In the applications section we also show performance metrics for the vortex particle method and panel method.

machine	*time* (sec)
Gravity	100,000 particles
Cray Y-MP	153.7 (1 proc)
Paragon	151.1 (1 proc)
Sparcstation/10	64.1 (1 proc)
CM-5	10.4 (32 proc)
Paragon	7.3 (32 proc)
Gravity	100,000 particles (isothermal)
Paragon	104.7 (1 proc)
SP-1	48.8 (1 proc)
Sparcstation/10	49.6 (1 proc)
Paragon	6.7 (32 proc)
SP-1	2.5 (32 proc)
Gravity	2,000,000 particles
CM-5	74.6 (64 proc)
Paragon	47.7 (64 proc)
SP-1	20.5 (64 proc)
CM-5E	10.8 (256 proc)
SPH+Gravity	100,000 particles
Paragon	592.8 (1 proc)
Sparcstation/10	242.7 (1 proc)
CM-5	33.4 (32 proc)
Paragon	23.5 (32 proc)

Table 1: A tabular comparison of several parallel and sequential machines for four different canonical test problems.

Global Potential Energy	-0.599706
Global PE error	5.58×10^{-4}
RMS PE Error	1.57×10^{-3}
Max PE Error	5.73×10^{-3}
RMS Force	0.774088
RMS Force Error	4.77×10^{-3}
Max Force Error	2.13×10^{-2}

Table 2: Errors for equal mass particles randomly distributed in a sphere with $R = M = 1$, using an error bound of 1% on each partial interaction.

A few details related to optimization should be mentioned. For the Paragon code, the $1/\sqrt{r^2}$ operation was coded in assembly language using a Newton-Raphson iteration. For the SP-1 code, the same operation was coded in `C` using a Chebychev polynomial approximation and one Newton iteration (see [20] for details). On the CM-5, no attempt was made to use the vector units. The YMP performance could possibly be improved a great deal by tuning of a few critical functions where vectorization was inhibited.

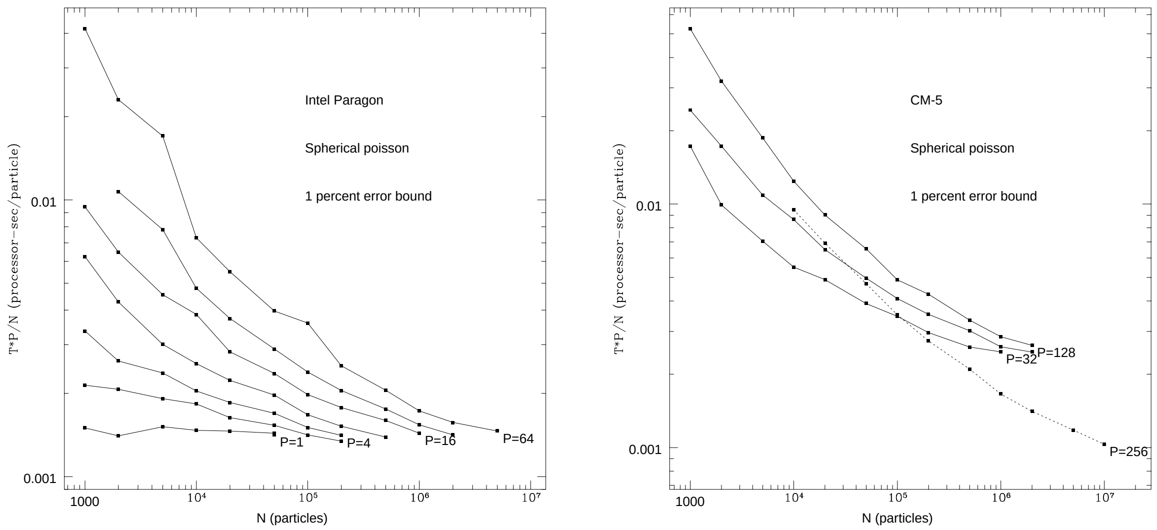

Figure 8: We plot performance results on the Intel Paragon and CM-5 for the $O(N)$ algorithm applied to a random spherical distribution. Note that the one processor results indicate $O(N)$ scaling. The dotted line in the CM-5 plot is for the CM-5E, that uses a faster SPARC processor.

9 Applications

Figure 9 shows a collision between a main-sequence star and a white dwarf star. This simulation was run on an Intel Paragon as a test problem to validate the correctness of our implementation of the SPH method. It took about 40 seconds per timestep on a 32 processor machine. This problem was previously used as the basis for a comparison between SPH and PPM in [21]. Figure 9 also shows a galaxy cluster simulation with 1.1 million particles computed using local expansions. This simulation of 2700 timesteps was completed in about 15 hours on 128 processors of an Intel Paragon. Additional applications of earlier versions of our treecode to large astrophysical N-body problems may be found in [22, 23] and [24].

9.1 The Vortex Particle Method

The vorticity equation ($\boldsymbol{\omega} = \nabla \times \mathbf{u}$, and hence $\nabla \cdot \boldsymbol{\omega} = 0$) for an incompressible fluid ($\nabla \cdot \mathbf{u} = 0$) is obtained from taking the curl of the momentum equation:

$$\frac{D\boldsymbol{\omega}}{Dt} = (\nabla \mathbf{u}) \cdot \boldsymbol{\omega} + \nu \, \nabla^2 \boldsymbol{\omega} \; , \tag{33}$$

where $\frac{Df}{Dt} = \frac{\partial f}{\partial t} + (\mathbf{u} \cdot \nabla) f$ is the Lagrangian derivative and ν is the kinematic viscosity. The vorticity equation is thus a nonlinear transport equation that can be solved using a particle method. The evaluation of the velocity field induced by a system of vortex particles is thus very similar to the evaluation of the acceleration field induced by a system of point masses in gravitation. One need only replace the scalar particle mass m^q with the vector vorticity strength, γ^q, and the real valued multiplication with a vector

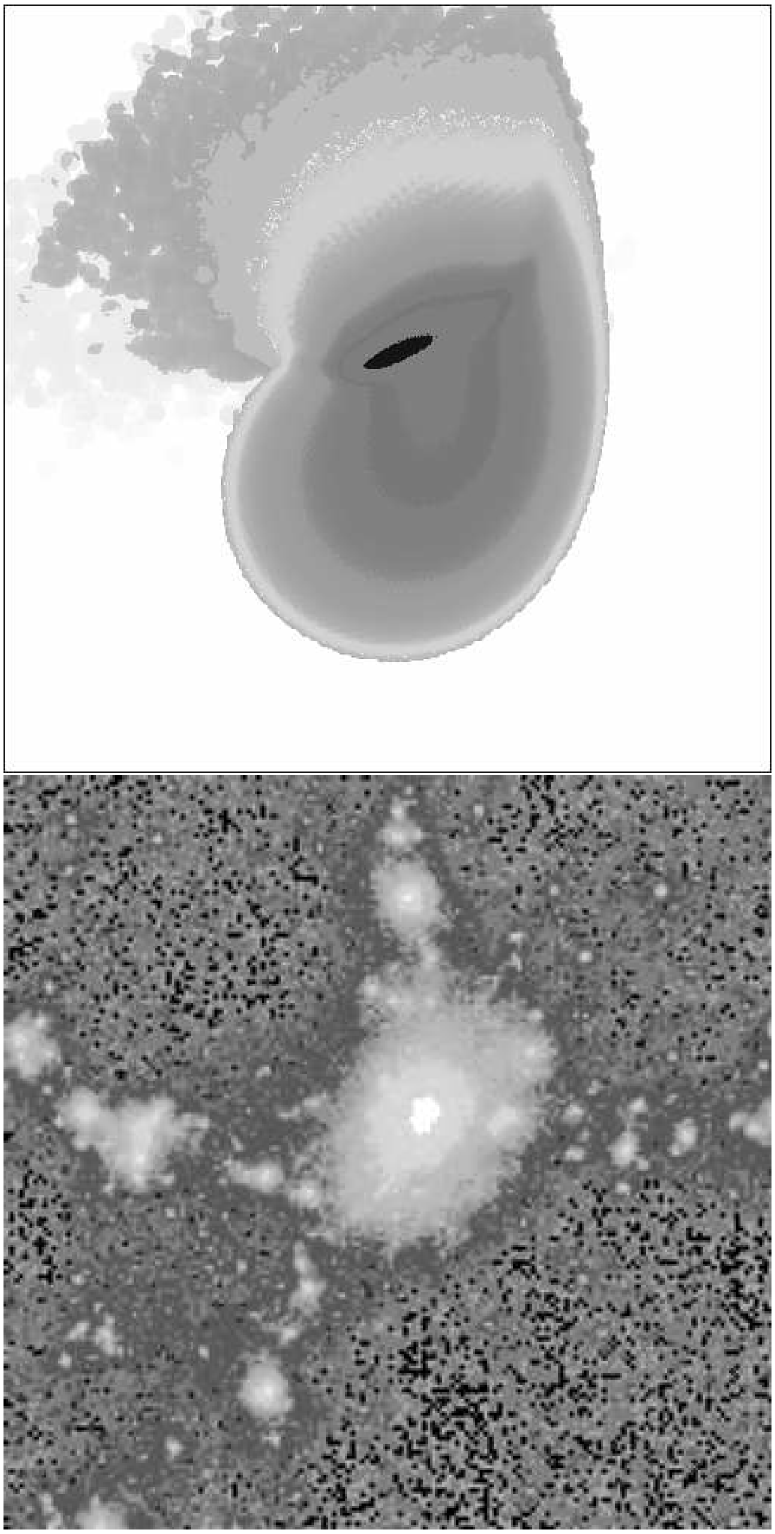

Figure 9: On the top is an intermediate stage of the collision between a 0.5 $M_{\odot}$ main-sequence star (modeled as an $n = 3/2$ polytrope) and a white dwarf star (modeled as a point mass). The encounter was simulated using SPH with 112,000 particles. On the bottom is the final state of a 1.1 million particle gravitational N-body simulation of the formation of a galaxy cluster.

cross-product. Just as with the gravitational case, it is possible to approximate the summation with an expression involving the multipole moments of the vorticity distribution. A more detailed description of the application of our treecode to this problem may be found in [25].

We carried out a series of timings for a problem representing the evolution of an initially spherical vorticity distribution. Figure 10 shows the initial positions of vortex particles representing a surface vorticity:

$$\boldsymbol{\mu} = \frac{3}{8\pi}\sin(\theta)\hat{\mathrm{e}}_{\varphi}. \tag{34}$$

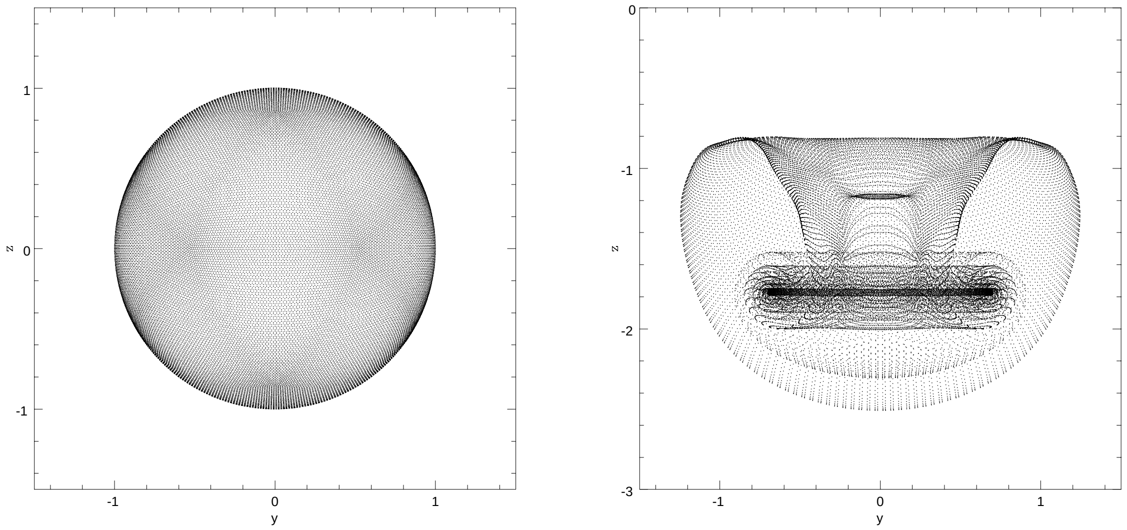

Figure 10: On the left are shown the positions of 81920 vortex particles initially on the surface of a unit sphere. Each of the particles carries a vector surface-vorticity, $\mu = \frac{3}{8\pi} sin(\theta)\hat{\mathrm{e}}_{\varphi}$ that is not shown for clarity. On the right is the result of evolving the initial state through 100 timesteps to t=2.5.

Timings are shown in Figure 11 for various $P = 1, 2, 4, \ldots 512$. The timings correspond to the total wallclock time per iteration, i.e., the time spent in parallel decomposition, computing the vector stream function $\boldsymbol{\psi}$, the velocity $\mathbf{u}$, and the gradient of $\mathbf{u}$ at each particle position, as well as updating the particle positions and strengths. The algorithm used in this case does not compute cell-cell interactions.

9.2 The Panel Method

For an inviscid, incompressible and irrotational fluid flow, the continuity equation can be reduced to the Laplace equation for the velocity potential,

$$\nabla^2 \boldsymbol{\Phi} = 0. \tag{35}$$

It is possible to superpose exact solutions of Eq. 35 in such a way that the boundary conditions are properly met. These so-called panel methods are used extensively for aerodynamics modeling in the aircraft and automobile industries [26]. Here we give a brief overview of the application of our treecode to the solution of this problem. A more in-depth analysis may be found in [27].

A distribution of panels in a uniform stream parallel to the x-axis produces a potential

$$\boldsymbol{\Phi}(\vec{r}_k) = U_\infty x_k + \frac{1}{4\pi}\sum_{j=1}^{N}\sigma_j \int_j \frac{1}{r_{kj}} ds_j \tag{36}$$

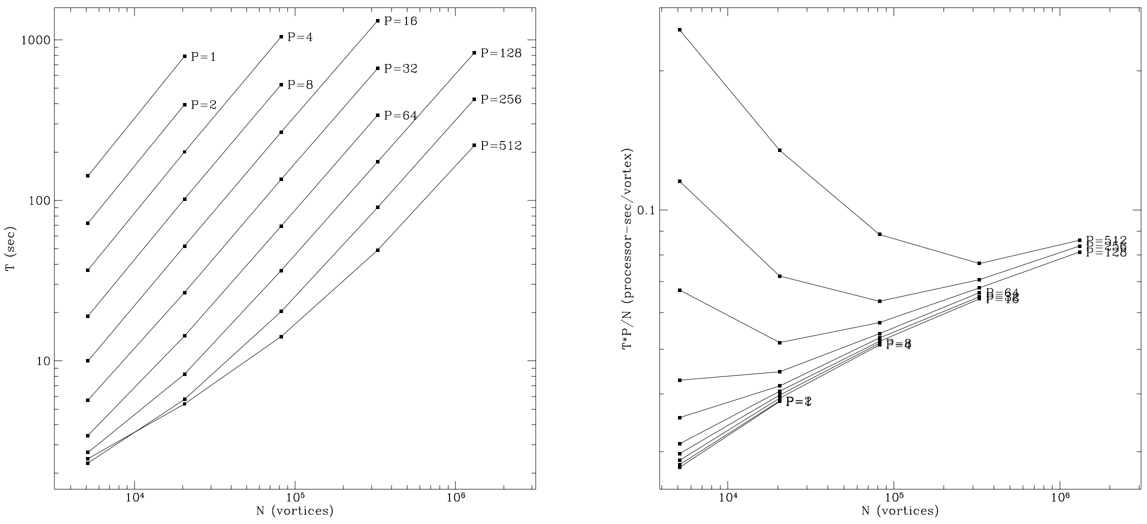

Figure 11: On the left, the time per timestep for the vortex method vs. number of vortices is shown, averaged over several timesteps. On the right is the same data replotted with an abscissa of $\frac{TP}{N}$, to better illustrate the dominant trends. Here, it is clear that one million bodies is in the "large-N" limit, and the parallel overhead (obtained by measuring the difference between the $P = 512$ curve and the extrapolation of the $P = 1$ curve) is in the neighborhood of 20%. The figure also demonstrates that the scaling behavior with N is again slightly super-linear. This time the exponent is approximately $T \propto N^{1.2}$.

where U_∞ is the velocity of the stream and σ_j is the source density of the jth panel. The problem is to determine σ_j to satisfy the boundary condition of no flow through the surface of the body. The boundary constraints for each control point produces a linear system of equations,

$$\boldsymbol{A\sigma} = \boldsymbol{R}. \tag{37}$$

This system of equations can be solved either directly in $O(N^3)$ time or iteratively in $MO(N^2)$ time, where M is the number of iterations required, which is both problem and method dependent. When evaluating the influence of the N panels in the system at the control point of panel p, it is possible to use a simplified formula if the panel q is far enough away. The improved method that we apply is to further combine multiple distant panels into a single multipole source, using the same fundamental idea as is used by a gravitational treecode.

We have solved two model potential flow problems around a unit sphere. Table 3 shows performance results for the panel code on the first very simple test problem: the potential flow past the unit sphere with uniform freestream. The exact solution for the velocity at the sphere surface is:

$$\mathbf{U}_{\text{exact}} = \frac{3}{2}\,\hat{\mathbf{e}}_n \times (\mathbf{U}_\infty \times \hat{\mathbf{e}}_n) = \frac{3}{2}\,(\mathbf{U}_\infty - (\mathbf{U}_\infty \cdot \hat{\mathbf{e}}_n)\,\hat{\mathbf{e}}_n) \tag{38}$$

where $\hat{\mathbf{e}}_n$ is the local unit normal. The problem is solved using a simple iterative solver: an underrelaxed Jacobi. The time reported in the table is the total time per iteration, i.e., the time necessary to complete the velocity evaluation for all panels.

N	e_{tol}	time(sec)
1280	1.0e-1	1.89
1280	1.0e-3	4.50
1280	1.0e-5	14.7
5120	1.0e-1	9.30
5120	1.0e-3	22.6
5120	1.0e-5	80.9
20480	1.0e-1	47.9
20480	1.0e-3	122
20480	1.0e-5	463
81920	1.0e-1	220
81920	1.0e-3	558
81920	1.0e-5	2310

Table 3: Performance results for the panel treecode: potential flow past the unit sphere with uniform freestream. $\|\mathbf{U}_{\text{exact}}\|_{\text{max}} = 1.50$. The first column is the number of panels used in the discretization of the sphere, and the second is the maximum allowed error in the gradient of the potential.

10 Conclusion

Particle methods are useful for studying an enormous variety of systems. Hockney and Eastwood [28] discuss applications in plasma physics, device physics, astrophysics and material science. Our code has allowed us to make significant progress in the study of galaxy dynamics [22] and cosmology [23]. In addition, we note applications in molecular dynamics [29], computational fluid dynamics [30, 31] and partial differential equations relevant to biology [32]. Fast multipole methods have been used to address two-dimensional problems in potential flows [33], and electromagnetic scattering [34]. Much chemistry is done with small N, and these may not benefit much from treecodes. Nevertheless, there are some important problems with larger N [35], which should run well with a parallel treecode.

Although we have described our methods in the context of particle data, treecodes are also useful in contexts completely outside dynamics. Oct-trees are a popular data structure in computer graphics and solid modeling [36]. Samet [17] discusses literally dozens of applications of quadtrees (the two-dimensional analog of oct-trees) for a variety of database related applications.

We have shown that treecodes can be successfully implemented on a variety of message passing parallel computers, and actually achieve the high performance that these machines promise. By careful implementation of a generic "tree" library, we are able to add new applications in a relatively painless manner. Tree-based codes can solve a very general class of problems that can be expected to grow in importance as the need for spatial adaptivity becomes necessary for the simulation of ever more difficult problems.

Acknowledgments

Special thanks to Melvyn Davies for much help with the SPH code, and G. S. Winckelmans for the implementation of the vortex and panel method modules. We would like to thank David Edelsohn for providing the benchmark results on the SP-1, and IBM Watson Research center for providing time on that machine. The authors wish to acknowledge the Advanced Computing Laboratory of Los Alamos National

Laboratory, Los Alamos, NM 87545; this work was performed in part on computing resources located at that facility. Time on the CM-5E was provided by the Naval Research Laboratory. This research was performed in part using the Intel Touchstone Delta System and the Intel Paragon operated by Caltech on behalf of the Concurrent Supercomputing Consortium. This research was supported in part by a grant from NASA under the HPCC program.

References

[1] D. Sugimoto, Y. Chikada, J. Makino, T. Ito, T. Ebisuzaki, and M. Umemura, "A special-purpose computer for gravitational many-body problems," *Nature*, vol. 345, p. 33, 1990.

[2] L. Lucy *Astron. J.*, vol. 82, p. 1013, 1977.

[3] R. A. Gingold and J. J. Monaghan, "Smoothed particle hydrodynamics: Theory and application to non-spherical stars," *M. N. R. A. S.*, vol. 181, p. 375, 1977.

[4] L. Hernquist and N. Katz, "Treesph: A unification of SPH with the hierarchical tree method," *Ap. J. Suppl.*, vol. 70, p. 419, 1989.

[5] W. Benz, R. L. Bowers, A. G. W. Cameron, and W. H. Press, "Dynamics mass exchange in doubly degenerate binaries. I. 0.9 and 1.2 $m_\odot$ stars," *Ap. J.*, vol. 348, p. 647, 1990.

[6] J. J. Monaghan, "Particle methods for hydrodynamics," *Computer Physics Reports*, vol. 3, p. 71, 1985.

[7] W. Benz in *Numerical Modeling of Stellar Pulsation* (J. R. Buchler, ed.), Dordrecht: Kluwer Academic, 1990.

[8] J. J. Monaghan, "Smoothed particle hydrodynamics," *Ann. Rev. Astron. Astrophys.*, vol. 30, p. 543, 1992.

[9] P. Laguna *Ap. J. (Letters)*, 1994. (submitted).

[10] J. K. Salmon, *Parallel Hierarchical N-body Methods*. PhD thesis, California Institute of Technology, 1990.

[11] J. K. Salmon and M. S. Warren, "Skeletons from the treecode closet," *J. Comp. Phys.*, vol. 111, pp. 136–155, 1994.

[12] J. E. Barnes, "An efficient N-body algorithm for a fine-grain parallel computer," in *The Use of Supercomputers in Stellar Dynamics* (P. Hut and S. McMillan, eds.), (New York), pp. 175–180, Springer-Verlag, 1986.

[13] J. Barnes and P. Hut, "A hierarchical O(NlogN) force-calculation algorithm," *Nature*, vol. 324, p. 446, 1986.

[14] M. S. Warren and J. K. Salmon, "Astrophysical N-body simulations using hierarchical tree data structures," in *Supercomputing '92*, (Los Alamitos), pp. 570–576, IEEE Comp. Soc., 1992.

[15] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy, "Load balancing and data locality in hierarchical N-body methods," *Journal of Parallel and Distributed Computing*, 1992. (in press).

[16] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree N-body algorithm," in *Supercomputing '93*, (Los Alamitos), pp. 12–21, IEEE Comp. Soc., 1993.

[17] H. Samet, *Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990.

[18] L. Greengard, *The Rapid Evalution of Potential Fields in Particle Systems*. PhD thesis, Yale University Computer Science, 1987.

[19] J. E. Barnes, "A modified tree code: Don't laugh; it runs," *J. Comp. Phys.*, vol. 87, no. 1, p. 161, 1990.

[20] A. H. Karp, "Speeding up N-body calculations on machines without hardware square root." (preprint), 1992.

[21] M. B. Davies, M. Ruffert, W. Benz, and E. Müller, "A comparison between SPH and PPM: Simulation of stellar collisions." (submitted), 1991.

[22] M. S. Warren, P. J. Quinn, J. K. Salmon, and W. H. Zurek, "Dark halos formed via dissipationless collapse: I. Shapes and alignment of angular momentum," *Ap. J.*, vol. 399, pp. 405–425, 1992.

[23] W. H. Zurek, P. J. Quinn, J. K. Salmon, and M. S. Warren, "Large scale structure after COBE: Peculiar velocities and correlations of cold dark matter halos," *Ap. J.*, vol. 431, p. 559, 1994.

[24] M. S. Warren, *Experimental Cosmology Using Fast Parallel N-body Methods*. PhD thesis, University of California, Santa Barbara, 1994.

[25] J. K. Salmon, M. S. Warren, and G. S. Winckelmans, "Fast parallel treecodes for gravitational and fluid dynamical N-body problems," *Intl. J. Supercomputer Appl.*, vol. 8, pp. 129–142, 1994.

[26] J. L. Hess, "Panel methods in computational fluid dynamics," *Ann. Rev. Fluid Mechanics*, vol. 22, p. 255, 1990.

[27] G. S. Winckelmans, J. K. Salmon, and M. S. Warren, "The fast solution of three-dimensional boundary integral equations in potential flow aerodynamics using parallel and sequential tree codes," 1994. (in preparation).

[28] R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*. New York: Mcgraw-Hill International, 1981.

[29] J. A. Board, J. W. Causey, J. F. Leathrum, A. Windemuth, and K. Schulten, "Accelerated molecular dynamics simulation with the parallel fast multipole algorithm," *Chem. Phys. Let.*, vol. 198, p. 89, 1992.

[30] A. Leonard, "Computing three-dimensional incompressible flows with vortex elements," *Ann. Rev. Fluid Mechanics*, vol. 17, p. 523, 1985.

[31] G. S. Winckelmans, *Topics in Vortex Methods for the Computation of Three- and Two- Dimensional Incompressible Unsteady Flows*. PhD thesis, California Institute of Technology, 1989.

[32] A. Sherman and M. Mascagni, "A gradient random walk method for two-dimensional reaction-diffusion equations." NIH Technical Report, 1991.

[33] L. Greengard, "Potential flow in channels," *SIAM J. Sci. Stat. Comp.*, vol. 11, no. 4, p. 603, 1990.

[34] N. Engheta, W. D. Murphy, V. Rokhlin, and M. S. Vassiliou, "The fast multipole method (FMM) for electromagnetic scattering problems," *IEEE Transactions on Antennas and Propagation*, vol. 40, no. 6, pp. 634–642, 1992.

[35] H.-Q. Ding, N. Karasawa, and W. Goddard, "Atomic level simulations of a million particles: The cell multipole method for coulomb and london interactions," *J. of Chemical Physics*, vol. 97, pp. 4309–4315, 1992.

[36] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics, Principles and Practice*, p. 550. Reading, MA: Addison-Wesley, 1990.