Parallel Tree Code Library for Vortex Dynamics

Pangfeng Liu
Department of Computer Science
National Chung Cheng University
Chia-Yi, Taiwan
pangfeng@cs.ccu.edu.tw

This paper describes the implementation of a parallel tree code library that can be used in the study of multi-filament vortex simulations. The simulations involve distributions that are irregular and time-varying. The library uses object-oriented techniques to isolate the tree structure details from the end users to enable fast prototyping of N-body tree codes. The library has been used previously to develop N-body code for gravitational force field computation, and in this paper we further demonstrate its versatility by implementing a vortex dynamic simulation code using Biot-Savart model. The additional advantages of the library include automatic parallelism and portability among different parallel platforms and we report very competitive timing results on Sun Ultra cluster.

1. Introduction

Computational methods to track the motions of bodies which interact with one another, and possibly subject to an external field as well, have been the subject of extensive research for centuries. So-called "N-body" methods have been applied to problems in astrophysics, semiconductor device simulation, molecular dynamics, plasma physics, and fluid mechanics.

Computing the field at a point involves summing the contribution from each of the $N-1$ particles. The direct method evaluates all pairs of two-body interactions. While this method is conceptually simple, vectorizes well, and is the algorithm of choice for many applications, its $O(N^2)$ arithmetic complexity rules it out for large-scale simulations involving millions of particles iterated over many time steps.

Larger simulations require faster methods involving fewer interactions to evaluate the field at any point. In the last decade a number of approximation algorithms have emerged; the fastest methods require only $O(N)$ evaluations [13]. All heuristic algorithms exploit the observation that the effect of a cluster of particles at a distant point can be approximated by a small number of initial terms of an appropriate power series. This leads to an obvious divide-and-conquer algorithm in which the particles are organized in a hierarchy of clusters so that the approximation can be applied efficiently. Barnes and Hut [4] applied this idea to gravitational simulations. More sophisticated schemes were developed by Greengard and Rokhlin [13] and subsequently refined by Zhao [28], Anderson [1], and better data structures have been developed by Callahan and Kosaraju [7].

Several parallel implementations of the N-body tree algorithms mentioned above have been developed. Salmon [24] implemented the Barnes-Hut algorithm on the NCUBE and Intel iPSC, Warren and Salmon [25] reported experiments on the 512-node Intel Touchstone Delta, and later developed hashed implementations of a global tree structure which they report in [26,27]. They have used their codes for astrophysical simulations and also for vortex dynamics. This paper builds on our previous CM-5 implementations of the Barnes-Hut algorithm for astrophysical simulation [5,20–22] and vortex dynamics [6,11], and contrast all the previous efforts with an easy-to-use object-oriented programming interface that provides automatic parallelism.

The remainder of this abstract is organized as follows. Section 2 describes the multi-filament vortex simulation in some detail. Section 3 outlines the fast summation algorithm which is essentially the Barnes-Hut algorithm. Section 4 describes the structure of the tree library and how we use it to build implicit global trees in distributed memory. Section 5 describes experimental results on Sun Ultra workstation cluster.

2. Vortex method in fluid dynamics

Many flows can be simulated by computing the evolution in time of vorticity using Biot-Savart models. Biot-Savart models offer the advantage that the calculation effort concentrates in the regions containing the vorticity, which are usually small compared to the domain that contains the flow. This not only reduces considerably the computational expense, but allows better resolution for high Reynolds number flows. Biot-Savart models also allow us to perform effectively inviscid computations with no accumulation of numerical diffusion [2,18,19].

Vortex method are also appropriate for studying the many complex flows that present coherent structures. These coherent structures frequently are closely interacting tube-like vortexes. Some of the flows of practical interest that present interacting tube-like vortex structures are wing tip vortices [9,16], a variety of 3D jet flows of different shapes [14,15] and turbulent flows [10].

The study of small scales formation in high Reynolds number flows tends to require substantial amount of computational resources. In multi-filament models, the $O(N^2)$ nature of computational expense of the Biot-Savart direct method (where N is the number of grid points) severely limits the vortex collapse simulations. leaving the most interesting cases of collapse beyond the cases that have been examined to date[12]. Therefore, fast simulation algorithms, like various tree codes in N-body simulation, should be used to increase the resolution under a given computation resource constraint.

2.1. The Biot-Savart model

The version of the vortex method we use was developed by Knio and Ghoniem [17]. The vorticity of a vortex tube is represented by a bundle of vortex filaments $\boldsymbol{\chi}_l(\sigma, t^*)$, each of them with circulation Γ_l. The n_f filaments forming the bundle are advected according to the velocity field

$$\mathbf{u}(\mathbf{x}) \;=\; -\sum_{l=1}^{n_f} \frac{\Gamma_l}{4\pi} \int_C \frac{(\mathbf{x}-\boldsymbol{\chi}_l)\times d\boldsymbol{\chi}_l}{|\mathbf{x}-\boldsymbol{\chi}_l|^3}\, g(|\mathbf{x}-\boldsymbol{\chi}_l|)\;, \tag{1}$$

where $g(\rho) \;= 1 \;-\; \exp(\,-\rho^3/\delta^3\,)$.

2.1.1. Discretization

Each filament of the vortex ring is discretized by n_0 grid points or vortex elements. Once this is done, the order of the summations in Equation 1 is unimportant, i.e. (1) is solved numerically at N discrete points or vortex elements $\boldsymbol{\chi}_p$ by using the approximation

$$\mathbf{u}(\mathbf{x}) \;=\; -\frac{1}{4\pi}\sum_{p=1}^{N_p} \frac{(\mathbf{x}-\boldsymbol{\chi}_p)\times\Delta\boldsymbol{\chi}_p\,\Gamma_p}{|\mathbf{x}-\boldsymbol{\chi}_p|^3}\, g(|\mathbf{x}-\boldsymbol{\chi}_p|)\;, \tag{2}$$

where the filament ordering has to be considered in the computation of the central difference $\Delta\boldsymbol{\chi}_j$

$$\Delta\boldsymbol{\chi}_j \;=\; \frac{1}{2}\,(\boldsymbol{\chi}_{j+1}-\boldsymbol{\chi}_{j-1})\;. \tag{3}$$

This is a characteristic of the filament approach in 3D vortex methods. In contrast with the "vortex arrow" approach [19,27], the evaluation of the vortex elements "strengths" in the filament method does not require the evaluation of the velocity gradient, which involves computing for another integral over all of the vortex element. Also, filaments with form of closed curves, satisfy the divergence free condition of the vorticity field. This is not always the case at all times in the vortex arrow approach.

The velocity field in eq. (2) can be computed more efficiently by using the multipole expansion

$$\mathbf{u}(\mathbf{x}) \;=\; -\sum_{n=0,j,k}^{m,j+k\le n} \boldsymbol{M}(j,k,n-j-k)\;\times\; D(j,k,n-j-k)\;\nabla\psi(\mathbf{x}-\boldsymbol{\chi}_0)\;+\;\nabla\times\boldsymbol{\Phi}_m\;, \tag{4}$$

where

$$\boldsymbol{M}(j,k,n-j-k) \;=\; \sum_{p=1}^{N_p} \boldsymbol{\alpha}_p\,(\chi_p-\chi_0)_1^j\,(\chi_p-\chi_0)_2^k\,(\chi_p-\chi_0)_3^{n-j-k} \tag{5}$$

and

$$D(j,k,n-j-k) \;=\; \frac{(-1)^n}{(n-j-k)!\;j!\;k!}\;\partial_1^j\,\partial_2^k\,\partial_3^{n-j-k}\;. \tag{6}$$

The strength of the vortex element is $\boldsymbol{\alpha}_p \;=\; \Gamma_p\,\Delta\boldsymbol{\chi}_p$ and $\psi(\rho) \;=\; \int_0^\rho g(s)\,ds/s^2 \;+\; \psi(0)$. The term $\nabla\times\boldsymbol{\Phi}_m$ is the truncation error that results from using a finite number of terms in the multipole expansion (4).

2.2. Initial conditions

The multi-filament ring is constructed around the center line

$$(\chi_1,\ \chi_2,\ \chi_3)\ =\ (a\ \cos\theta,\ b\ \sin\theta,\ c\ \sin 2\theta)\ , \tag{7}$$

where $0 \leq \theta \leq 2\pi$. The grid points are located at equally spaced intervals $\Delta\theta$ or at variable intervals, with the smaller intervals on the collapse region. We call this geometry the "Lissajous-elliptic" ring because of its projections on two orthogonal planes (for $c > 0$). The thickness of the multi-filament ring is δ_C. The circulation distribution Γ_i, and the initial filament core radius δ_i also need to be specified. Besides the fact that its parameter space contains cases of very rapid collapse, the low number of parameters of this configuration allows a complete parameter study at less computational expense. Low aspect ratio elliptic rings $a > b, c = 0$ correspond to rings with periodic behavior that can be used for dynamic and long time behavior testing of the algorithm.

3. N-body problem and Barnes-Hut algorithm

To reduce the complexity of computing the sum in Equation 2, we use the Barnes-Hut algorithm. The Barnes-Hut algorithm is one of the "tree codes" that all explore the idea that the effect of a cluster of vortex elements at a distant point can be approximated by a small number of initial terms of a Taylor series (Equation 4). To apply the approximation effectively, these tree codes organize the bodies into a hierarchy tree in which a vortex element can easily find the appropriate clusters for approximation purpose.

The Barnes-Hut algorithm proceeds by first computing an oct-tree partition of the three-dimensional box (region of space) enclosing the set of vortex elements. The partition is computed recursively by dividing the original box into eight octants of equal volume until each undivided box contains exactly one vortex element[1]. An example of such a recursive partition in two dimensions and the corresponding BH-tree are shown in Figure 1. Note that each internal node of the BH-tree represents a cluster. Once the BH-tree has been built, the multipole moments of the internal nodes (Equation 5) are computed in one phase up the tree, starting from the leaves.

To compute the new velocity, we loop over the set of vortex elements observing the following rules. Each vortex element starts at the root of the BH-tree, and traverses down the tree trying to find clusters that it can apply approximation. If the distance between the vortex element and the cluster is far enough, with respect to the radius of the cluster, then the velocity due to that cluster is approximated by a single interaction between the vortex element and the multipoles located at the geometry center of the cluster. Otherwise the vortex element visits each of the children of the cluster.[2] The leaves of the subtree traversed by a vortex element will be called *essential data* for the vortex element because

[1]In practice it is more efficient to truncate each branch when the number of vortex elements in its subtree decreases below a certain fixed bound.

[2]Formally, if the distance between a vortex element and a cluster is more than RADIUS (cluster)$/\theta$, then we will approximate the effect of that cluster as a point mass.

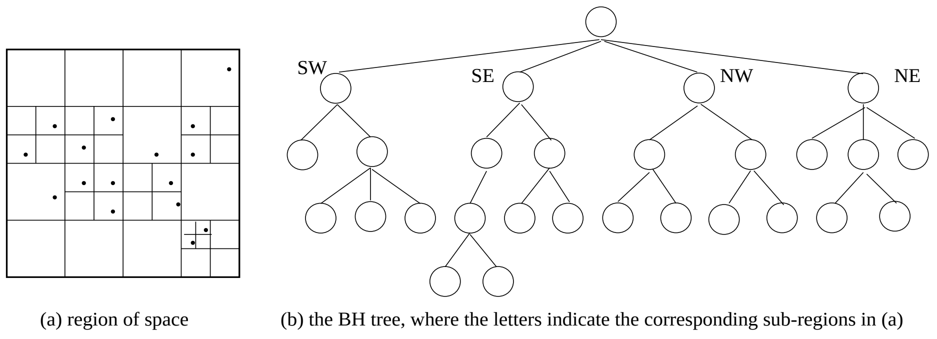
SW
SE
NW
NE
(a) region of space
(b) the BH tree, where the letters indicate the corresponding sub-regions in (a)

Figure 1. BH tree decomposition.

it needs these nodes for interaction computation. Once the velocity on all the vortex elements are known, the new positions can be computed.

4. Tree Code Library

Figure 2 illustrates the class hierarchy in the tree code library. The *generic tree layer* supports simple tree construction and manipulation methods. The *Barnes-Hut tree layer* extends *generic tree layer* (Sec 4.2) to include BH algorithm specific operations. The *application layer* uses classes in the *Barnes-Hut tree layer*, to develop applications.

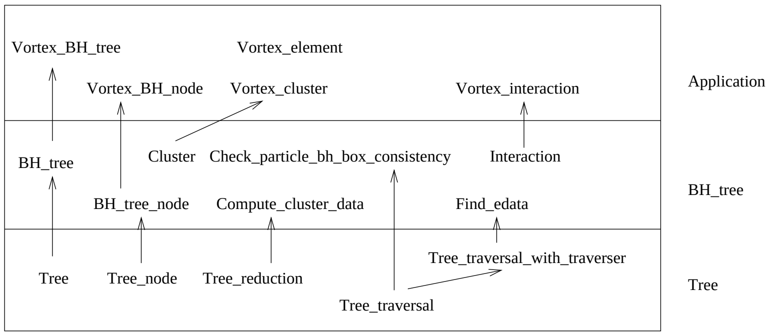

Figure 2. The class hierarchy in *generic tree*, *Barnes-Hut tree*, and *application* layers.

4.1. Generic tree layer

The *generic tree layer* is the foundation of our framework from which complex tree structures can be derived. The class `Tree` serves as a container class in which every tree node has a pointer to data of the given data type. The desired data type is given as a template parameter. We define basic tree manipulation methods in the generic tree layer, including inserting/deleting a leaf from the tree, and performing tree reduction and traversal.

We have also implemented two tree operations – *reduction* and *traversal*, as special classes. Objects instantiated from the `reduction` class compute the data of a tree node according to the data of its children, e.g. computing the multipole moments in Equation 5. Objects instantiated from the `traversal` class walk over the tree nodes, and perform a user-defined operation (denoted as *per node function*) on each tree node.

The tree reduction/traversal operations were implemented in an application-independent manner. Both operations are implemented as class templates so that users can supply tree and tree node type for a customized tree reduction/traversal operations. For tree reduction, users are required to provide two functions: `init(Data*)` and `combine(Data *parent, Data* child)`, which tell `reduction` class how to initialize and combine the data in tree nodes, respectively. For tree traversal, users are required to provide the per node function `bool process(Data*)` that is to be performed on every tree node.

4.2. Barnes-Hut tree

By extending the `Tree` class, each tree node in `BH_tree` contains a data cluster, and the data cluster of each leaf node contains a list of bodies.[3] The types of the body and cluster are given by the user of the `BH_tree` class as template parameters `AppCluster` and `AppBody`. The `BH_tree` class also supports several operations: computing cluster data, finding essential data, computing interaction, and checking body position and BH box for consistency. Most of these methods can be reused in implementing the fast multipole method.

Cluster data computation is implemented as a tree reduction. `init(AppCluster* cluster)` resets the data in the cluster and if the cluster is a leaf, it combines the data of the bodies from the body list into the data of the cluster. The other function `combine(AppCluster* parent, AppCluster* child)` adds children's data to parent's.

After collecting the essential clusters and bodies, a body can start computing the interactions. The computation class `Interaction` goes through the essential data list[4] and calls for functions to compute body-to-body and body-to-cluster interactions defined by the user of `Interaction`.

After bodies are moved to their new positions, they may not be in their original BH boxes. Therefore, the tree structure must be modified so that it becomes consistent with the new body positions again. This function is universally useful for all tree code because the dynamic tree structure is expensive to rebuild, but relatively cheap to patch up.

4.3. Application Layer

We now show an example of application development using the tree framework – a vortex dynamic N-body application built on top of the `BH_tree` layer. First we construct

[3]Recall that each leaf may have more than one body.

[4]Lists obtained from the class `Find_Edata`.

a class `Vortex_element` for bodies that interact with one another according to Equation 2, then we build the cluster type `Vortex_cluster` from `Vortex_element`. Next, in the `Vortex_cluster` class we define the methods for computing/combining moments and the methods for testing essential data.

Then, in class `Vortex_interaction`, which is derived from the class template `Interaction`, we define methods to compute vortex element interactions. We specify the vortex element interaction rules in the definition of `body_body_interaction` and `body_cluster_interaction`.

Finally, we define the BH-tree type `Vortex_BH_tree` and tree node type `Vortex_BH_node`. These two data types serve as template parameters to instantiate BH-tree related operations, like `Compute_cluster_data`, `Find_edata`, and `Check_particle_bh_box_consistency`. See Figure 2 for their inheritance relations.

4.4. Parallel implementation

In our current implementation, we assume SPMD (single program multiple data) model for parallel computation. Under this model, we would require abstractions for data mapping and interprocessor communication. We have designed two groups of classes for this purpose – `Mapper` classes that are responsible for defining the geometry of the tree structure, and `Communicator` classes that provide all-to-some communications that are common in N-body simulations.

Note that although using the same name, our Communicator class is quite different from the communication package in [8]. Our Communicator is a C++ class with a high level communication protocol. Therefore, the only optimizations we perform is message aggregation and random destination permutation, and leave all the other optimization to MPI library. Whereas the Communicator in [8] is a low level optimizer that will remove redundant communication, combining separate communications, and perform communication pipelining automatically.

Mapper classes

The `Mapper` classes define the geometry of data structures (e.g. BH trees in N-body simulations). Over the course of a simulation, `Mapper` objects are created during the construction of data structure objects (e.g. BH tree objects). When created, a `Mapper` object invokes the data partitioning function specified by the user or performs default behavior when no partitioning strategy is specified, it then gathers and caches geometry information from the partitioning function. In later stage of a simulation, the `Mappers` mediate object operations that require interprocessor communication.

In our previous parallel C implementation, we constructed an ORB partitioner and two associated geometry resolution functions: `data_to_processor` (that translates a data coordinate to a processor domain) and `dataset_to_processors` (that translates a rectangular box, which contains multiple data, to a set of processor domains). In addition, we defined a simple data structure `MappingTable` to store the ORB map. These data and methods have been integrated into the `Mapper` classes in our parallel framework. As part of this research effort, we are also extending the `Mapper` class to incorporate a number of commonly used partitioning strategies and user-defined mapping methods.

Communicator classes

The `Communicator` classes support general-purpose all-to-some communications for N-body tree codes. A `Communicator` class defines two functions: `extract` (that, when given a data pointer, constructs an outgoing data) and `process` (that processes each incoming data). When a `communicator` is constructed, it goes over the list of data pointers, calls `extract` to build outgoing data, packs many outgoing data into actual messages, sends/receives all the messages according to the communication protocol, and finally unpacks messages and calls `process` to perform appropriate actions.

The technique we developed for `communicator` has proven to be both efficient and general enough to support all-to-some communication in N-body tree codes. For instance, the essential data gathering was implemented as a tree traversal followed by a communicator phase. The tree traversal goes over the BH nodes, computes the proper destination set where the tree node might be essential, and appends its address to a pointer list to that destination. Each destination processor will have a separate pointer list that contains the addresses of those tree nodes that might be essential to the destination's local vortex elements. The `extract` routine assures that only essential parts of a tree node are transmitted. The `process` routine inserts incoming data into the local tree. All the message packing/unpacking/transmission are handled by `communicator`.

5. Experimental Study

We demonstrate the flexibility of the parallel tree library by implementing a multi-filament fluid dynamic calculation code, in addition to the previous gravitational force field computation code in [23]. Both applications were developed within the tree library framework; therefore, all the tree structure details and communications were taken care of by predefined tree operations and the communicator classes.

The experiments were conducted on four UltraSPARC-II workstations located in the Institute of Information Science, Academia Sinica. The workstations are connected by a fast Ethernet network capable of 100M bps per node. Each workstation has 128 mega bytes of memory and runs SUNOS 5.5.1. The communication library in the framework is implemented on top of MPI (mpich version 1.0.4 [3]).

The multiple-filament vortex method computes the vorticity on each vortex element, and requires an extra phase in the tree construction to compute the vorticity. The vorticity of a vortex element is defined as the displacement of its two neighbors in the filament (Equation 3). Once the vorticity on each vortex element is computed, we can compute the multipole moments on the local trees. Finally, each processor sends its contribution to a node to the owner of the node so that individual contributions are combined into globally correct information, as in the gravitational case [23].

The fluid dynamics code developed using the tree framework delivers competitive performance. The speedup factors are higher than those of the gravitation code [23] because the fluid dynamics code performs more computation on each vortex element, which amortizes the overhead of parallelization and object orientation.

problem size	8k	16k	24k	32k	40k	48k	56k	64k	128k	256k
sequential time	17.38	41.78	67.53	93.75	122.23	148.60	175.46	204.18	404.34	801.81
parallel time	5.51	12.81	19.77	27.84	34.96	43.00	51.37	59.05	117.20	231.07
speedup	3.15	3.26	3.42	3.38	3.46	3.42	3.42	3.46	3.45	3.47

Table 1
Timing comparison for the fluid codes. Time units are seconds. The parallel code were written using the tree framework, and the sequential code was converted from Barnes and Hut's code.

Acknowledge

The author thank Norman Zabusky and Victor Fernadez of Rutgers University for providing all the numerical details in the vortex dynamic simulations in Section 2, and Marina Chen of Boston University for motivating the idea of a reusable tree code library, and Sandeep Bhatt of Bell Communication Research for many helpful discussions during the early stage of this project. This work is supported by National Science Council grant 87-2213-E-194-008 and 86-2213-E-001-010.

REFERENCES

1. C. Anderson. An implementation of the fast multipole method without multipoles. *SIAM Journal on Scientific and Statistical Computing*, 13, 1992.
2. C. Anderson and C. Greengard. The vortex merger problem at infinite Reynolds number. *Communications on Pure and Applied Mathematics*, XLII:1123–1139, 1989.
3. Argonne National Labortory and University of Chicago. *MPICH, a protable implementation of MPI*, 1996.
4. J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324, 1986.
5. S. Bhatt, M. Chen, C. Lin, and P. Liu. Abstractions for parallel N-body simulation. In *Scalable High Performance Computing Conference SHPCC-92*, 1992.
6. S. Bhatt, P. Liu, V. Fernadez, and N. Zabusky. Tree codes for vortex dynamics. In *International Parallel Processing Symposium*, 1995.
7. P. Callahan and S. Kosaraju. A decomposition of multi-dimension point-sets with applications to k-nearest-neighbors and N-body potential fields. *24th Annual ACM Symposium on Theory of Computing*, 1992.
8. S.E. Choi and L.Snyder. Quantifying the effects of communication optimizations. In *Proceedings of the International Conference on Parallel Processing*, 1997.
9. S. C. Crow. Stability theory for a pair of trailing vortices. *AIAA Journal*, 8(12):2172–2179, 1970.

10. S. Douady, Y. Couder, and M. E. Brachet. Direct observation of the intermittency of intense vorticity filaments in turbulence. *Physical Review Letters*, 67(8):983–986, 1991.
11. V. Fernadez, N. Zabusky, S. Bhatt, P. Liu, and A. Gerasoulis. Filament surgery and temporal grid adaptivity extensions to a parallel tree code for simulation and diagnostics in 3d vortex dynamics. In *Second International Workshop in Vortex Flow*, 1995.
12. V.M. Fernandez, N.J. Zabusky, and V.M. Gryanik. Near-singular collapse and local intensification of a "Lissajous-elliptic" vortex ring: Nonmonotonic behavior and zero-approaching local energy densities. *Physical of Fluids A*, 6(7):2242–2244, 1994.
13. L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73, 1987.
14. H. S. Hussain and F. Hussain. Elliptic jets Part 1. Characteristics of unexcited and excited jets. *Journal of Fluid Mechanics*, 208:257–320, 1989.
15. H. S. Hussain and F. Hussain. Elliptic jets Part 3. Dynamics of preferred mode coherent structure. *Journal of Fluid Mechanics*, 248:315–361, 1993.
16. R. T. Johnston and J. P. Sullivan. A flow visualization study of the interaction between a helical vortex and a line vortex. Submitted to Experiments in Fluids, 1993.
17. O. M. Knio and A. F. Ghoniem. Numerical study of a three-dimensional vortex method. *Journal of Computational Physics*, 86:75–106, 1990.
18. A. Leonard. Vortex methods for flow simulation. *Journal of Computational Physics*, 37:289–335, 1980.
19. A. Leonard. Computing three-dimensional incompressible flows with vortex elements. *Annu. Rev. Fluid Mech.*, 17:523–559, 1985.
20. P. Liu. *The parallel implementation of N-Body algorithms.* PhD thesis, Yale University, 1994.
21. P. Liu and S. Bhatt. Experiences with parallel n-body simulation. In *6th Annual ACM Symposium on Parallel Algorithms and Architecture*, 1994.
22. P. Liu and S. Bhatt. A framework for parallel n-body simulations. In *Third International Conference on Computational Physics*, 1995.
23. P. Liu and J. Wu. A framework for parallel tree-based scientific simulation. In *26th International Conference on Parallel Processing*, 1997.
24. J. Salmon. *Parallel Hierarchical N-body Methods.* PhD thesis, Caltech, 1990.
25. M. Warren and J. Salmon. Astrophysical N-body simulations using hierarchical tree data structures. In *Proceedings of Supercomputing*, 1992.
26. M. Warren and J. Salmon. A parallel hashed oct-tree N-body algorithm. In *Proceedings of Supercomputing*, 1993.
27. M. Warren, J. Salmon, and G. Winckelmans. Fast parallel tree codes for gravitational and fluid dynamical N-body problems. *Intl. J. Supercomputer Applications*, 8.2, 1994.
28. F. Zhao. An $O(N)$ algorithm for three dimensional N-body simulation. Technical report, MIT, 1987.