JOURNAL OF COMPUTATIONAL PHYSICS **87**, 137–147 (1990)

Vectorization of Tree Traversals

LARS HERNQUIST

The Institute for Advanced Study, Princeton University, Princeton, New Jersey

Received July 18, 1988; revised April 4, 1989

A simple method for vectorizing tree searches, which operates by processing all relevant nodes at the same depth in the tree simultaneously, is described. This procedure appears to be general, assuming that gather–scatter operations are vectorizable, but is most efficient if the traversals proceed monotonically from the root to the leaves, or *vice versa*. Particular application is made to the hierarchical tree approach for computing the self-consistent interaction of N bodies. It is demonstrated that full vectorization of the requisite tree searches is feasible, resulting in a factor $\sim$4–5 improvement in cpu efficiency in the traversals on a CRAY X-MP. The overall gain in the case of the Barnes–Hut tree code algorithm is a factor $\sim$2–3, implying a net speed-up of $\approx$400–500 on a CRAY X-MP over a VAX 11/780 or SUN 3/50 © 1990 Academic Press, Inc.

I. INTRODUCTION

The use of tree-structured data to store and process information is wide-spread in computer science. In an abstract sense, trees provide a natural means for representing sets whose elements have a special relation to one another. This is especially true for hierarchical sets, in which some elements are ranked above others, where trees allow a simple visualization of any substructure that might be present. Trees can also be used to process elements that are linearly ordered, providing the basis for efficient searching and sorting algorithms. As a final example, trees are frequently used in decision-making, particularly if a given choice leads to many other alternatives.

Recently, tree structures have been successfully applied to the problem of simulating dynamical physical systems using particles. If long-range forces are present, trees can be used to efficiently compute the influence of remote groups of particles using multipole expansions. In the "hierarchical tree" approach to N-body dynamics (e.g., [1–5]) trees are used to reduce the operations count per step from $\sim O(N^2)$, as would be required by a simple direct sum, to $\sim O(N \log N)$, without introducing a fixed grid, as in particle–mesh methods. A more refined technique, known as the "fast multipole method" (e.g., [6–8]), also relies on trees to perform multipole expansions, but with a scaling $\sim O(N)$.

Trees are also useful for computing short-range interactions. For example, in particle-hydrodynamic methods, local forces, such as those arising from pressure gradients, contribute to the equations of motion. In "smoothed particle hydrodynamics" (e.g., [9]) these terms are estimated from the local distribution of particles. Consequently, it is desirable to perform nearest neighbor searching efficiently. Recently, Benz [10], Hernquist [11], and Hernquist and Katz [12] have discussed the merits of tree-based neighbor detection and this approach appears to be promising if the search interval is variable. Similar physical settings include screened plasmas, molecular dynamics, and nuclear interactions (e.g., [7, 13]).

In all these applications tree traversals can be used to identify the interactions among the particles. Intuitively, these operations appear to be most well suited for parallel computers such as the Connection Machine (e.g., [14]). For a vector architecture, as a CRAY X-MP or CRAY-2, the optimal procedure has not yet been established and, indeed, most previous implementations of tree-based particle schemes have processed trees in scalar mode (e.g., [10, 11, 15]). Since the cpu usage is dominated by scalar arithmetic, in such cases, there is strong motivation for a vectorization of the requisite tree searches.

In this paper, I describe one approach to the vectorization of tree traversals. Rather than processing trees *node* by *node*, as in a scalar implementation, this algorithm traverses the trees *level* by *level*. Vectorization is achieved by looping over all relevant nodes, at the same level, simultaneously. This method is not unique (e.g., [16, 17]), but has the virtue of preserving the basic structure of the scalar algorithm, which is an advantage if multiple time-scales are present (e.g., [12]).

In Section II, I motivate the need for an efficient tree traversal procedure by summarizing the properties of the hierarchical tree method. The vectorized algorithm is described in Section III, along with simple timing tests. Finally, a summary is given in Section IV.

II. The Hierarchical Tree Method

(a) *Basic Principles*

The straightforward particle–particle (PP) technique for computing long-range interactions, in which the force is determined by direct summation, has a number of advantages over other potential solvers. In particular, it is fully Lagrangian and does not use a grid. However, since the computing time per step scales as $\sim O(N^2)$ this approach is prohibitively expensive for large N.

Recently, a new class of N-body algorithms has been proposed which retains many of the advantages of the PP technique (e.g., [1–5]). Rather than compromising the spatial resolution and/or imposing geometrical restrictions, the hierarchical tree method introduces explicit approximations into the calculation of the potential to improve efficiency.

Particles are first organized into a nested hierarchy of clusters or cells and multipole moments of each cluster or cell are computed up to a fixed order. The acceleration is obtained by allowing each individual particle to interact with various elements of the hierarchy, subject to a prescribed accuracy criterion. As a rule, the force from nearby particles is computed by direct summation. The influence of remote particles is included by evaluating the multipole expansions of the clusters or cells which satisfy the accuracy requirement at the location of each particle. Typically, the number of terms in the expansions is small compared with the number of particles in the corresponding cluster or cell, and a significant gain in efficiency is realized.

If the accuracy criterion is such that the size of an unresolved cluster or cell increases in proportion to the distance from a given point, then a sum over N particles is replaced by a sum over log N interactions, and the cost per step scales as $\sim O(N \log N)$. The cpu efficiency can be further improved by symmetrizing the force calculation between clusters or cells. In the *fast multipole method* [6–8, 18] particles are organized into cells and cell–cell interactions are computed prior to the force calculation. Once these have been determined the force on a single particle can be obtained in a time independent of N, resulting in an overall scaling $\sim O(N)$.

The construction and maintenance of the hierarchy and subsequent determination of interactions can be performed efficiently through the use of tree-structured data (e.g., [1–5, 7, 15, 19–21]). Unlike fixed grids, tree structures are adaptive and do not constrain the global geometry or local spatial resolution. Thus, the hierarchical tree method retains most of the advantages of the PP technique, but at a significant reduction in computing costs.

(b) *The Barnes–Hut Algorithm*

The Barnes–Hut method relies on a hierarchical subdivision of space into regular cubic cells. At each step, prior to the force evaluation, a tree structure is built to store the hierarchy. Each node in the tree is associated with a cubic volume of space containing a given number of particles. In 3D, each volume is subdivided into eight subunits of equal volume, which become the descendents of the original node in the tree structure. This process is continued recursively until each fundamental subcell contains either one or zero particles. Empty cells are not stored explicitly in the tree, thus the leaves represent volumes of space containing precisely one particle. As part of the tree-building procedure, the total mass, center of mass coordinates, and quadrupole moments of each cell are computed recursively.

The force on a given particle is determined by walking through the tree, beginning at the top of the hierarchy (i.e., largest volume). At each step, the size of the current cell, s, is compared with its distance from the particle, d. If

$$\frac{s}{d} \leqslant \theta, \qquad (1)$$

where θ is a fixed tolerance parameter, then the influence of all particles within the cell is computed as a single particle–cell interaction. Otherwise, the cell is subdivided by continuing the descent through the tree until either the tolerance criterion is satisfied or an elementary cell is reached. In this manner, all operations, including tree construction and force evaluation can be performed on $O(N \log N)$ time.

III. Vectorization

(a) *Background*

It is conceptually useful to regard the tree-walk used to compute long-range forces as a device for establishing a list of interactions, i.e., a collection of cells and particles with which a given particle interacts. In Hernquist's [15] implementation of the Barnes–Hut hierarchical tree algorithm, which was designed for vectorizing supercomputers, interaction lists were established for single particles by walking through the tree from node to node. Given an interaction list, the subsequent force calculation was vectorized and an overall factor ~ 3–4 improvement in efficiency was realized from vectorization and compiler optimization. However, the cpu usage was dominated by the tree descent which was not at all vectorized.

Contrary to popular belief, tree descents can be fully vectorized. All algorithms proposed thus far rely on the possibility of vectorizing gather and scatter operations, i.e., indirect addressing. This feature was not available to Hernquist [15] on the MFECC CRAY X-MP, but exists on newer X-MPs as well as many other supercomputers.

Makino [17] succeeded in vectorizing all aspects of the hierarchical tree algorithm by performing tree descents for many particles simultaneously, vectorizing loops over particles. This scheme allows for the processing of long vectors and on machines such as CYBER 205s overall factors ~ 5 improvement in performance can be obtained.

A device for reducing the number of tree searches has been implemented by Barnes [16], who establishes interaction lists for *groups* of particles which are close together in space; a procedure commonly used in the fast multipole method [6–8, 18]. This shifts the bulk of the computation from the tree descents to force summation, by minimizing the required number of tree traversals. A further gain can be realized by combining the Makino and Barnes procedures and performing tree searches for many *groups* simultaneously (e.f., [12]).

A disadvantage of both of these approaches lies in the fact that they do not preserve the basic structure of the scalar version of the hierarchical tree method. That is, interaction lists are established for many particles simultaneously. This can be a hindrance if the force must be computed particle by particle, as is the case if each particle has a unique time step.

Here, I describe yet another technique for vectorizing tree traversals, which retains the basic structure of the original algorithm. Particles are processed individually, but the walk through the tree no longer proceeds node by node but rather level by level. That is, the tolerance criterion, Eq. (1), is simultaneously applied to all relevant cells at a given level. Those cells at the current level satisfying the accuracy requirements are added to the list of interactions. The remainder are subdivided and their descendents are placed on the list of cells to be visited on the next level further down.

The vectors which result from this procedure are typically short, and hence this approach will probably not be optimal for a CYBER 205. Empirically, the average vector length is rougly given by $\langle L_{\text{vector}} \rangle \approx L_0 \theta^{-2} \log_{10} N$, where L_0 is a constant depending on the density profile. For example, in the case of equilibrium Plummer models [15], $L_0 \approx 8$. Thus, for $N = 32768$ and $\theta = 0.6$, $\langle L_{\text{vector}} \rangle \approx 100$. However, the amount of overhead required is negligible and the tree descent routines are sped up by a factor of $\approx$4–5 on a CRAY X-MP, resulting in an overall gain of $\approx$2–3, depending on the number of terms retained in the multipole expansions.

(b) *A Vectorized Algorithm to Establish Interaction Lists*

By traversing trees level by level, the subroutine to establish interaction lists can be written concisely in standard, non-recursive FORTRAN. For example, suppose that it is desired to determine the interaction list for body *p*, using the tolerance criterion (1). In a form optimized for a CRAY, this subroutine might be written schematically as follows:

```
      SUBROUTINE treewalk(p, nterms, terms)
      INCLUDE "treedefs.h"
      INTEGER asubp(maxnnode), i, isubset(maxnnode), keepnear(maxnnode),
    &         nkeep, nnodes, node(maxnnode), nsubdiv, nterms, p, terms(maxnterm)
      LOGICAL tolcrit(maxnnode)
      nterms = 0
      nnodes = 1
      node(1) = root
 10   CONTINUE
      IF(nnodes. GT.0) THEN
        DO 20 i = 1, nnodes
          tolcrit(i) = ((pos(p, 1)-pos(node(i), 1))**2 + (pos(p, 2)-pos(node(i), 2))**2 +
    &           (pos(p, 3)-pos(node(i), 3))**2). GE. sizetol2(node(i))
          keepnear(i) = CVMGT(1, 0, tolcrit(i))
 20     CONTINUE
        CALL WHENEQ(nnodes, keepnear, 1, 1, isubset, nkeep)
        DO 30 i = 1, nkeep
          terms(nterms + i) = node(isubset(i))
 30     CONTINUE
        nterms = nterms + nkeep
        CALL WHENEQ(nnodes, keepnear, 1, 0, isubset, nsubdiv)
        DO 50 j = 1, 8
          DO 40 i = 1, nsubdiv
          asubp(i + (j - 1)*nsubdiv) = subp(node(isubset(i)), j)
 40       CONTINUE
 50     CONTINUE
        CALL WHENNE(8*nsubdiv, asubp, 1, 0, isubset, nnodes)
        DO 60 i = 1, nnodes
          node(i) = asubp(isubset(i))
 60     CONTINUE
        GO TO 10
      ENDIF
      RETURN
      END
```

Here, the argument *nterms* returns the number of bodies and cells on the interaction list, *terms*. The parameters *maxnterm* and *maxnnode*, defined in the include file *treedefs.h*, which is not shown, limit the lengths of the interaction list and the node arrays, respectively. The local variable *nnodes* is the number of nodes to examine at the current level, the indices of which are stored in the vector *node*.

The subroutine *treewalk* begins by putting the root node on the list of nodes to examine. It then executes the IF–THEN block following the continuation line 10 once for each level in the tree, until no further nodes need be checked. The loop 20 computes the tolerance criterion (1) for each node on the list, by comparing the square of the distance from body p to the node currently under scrutiny, *node*(i). The global variable *sizetol2* is the square of the linear size of cell *node*(i) divided by the square of θ. For bodies, $sizetol2 = 0$, and hence the test is, in this case, always satisfied. The intrinsic CRAY function CVMGT assigns either the value 1 or 0 to element i of *keepnear*, depending upon the outcome of the tolerance check.

Given the results of the test in the loop 20, *treewalk* then adds those terms satisfying the tolerance criterion to the interaction list in loop 30. The subroutine WHENEQ, which is CRAY-intrinsic, is used to select the elements of *keepnear* equal to 1. In fact, the intermediate variable *keepnear* would not even be needed if there existed a CRAY function to select elements of a logical vector. The second call to WHENEQ identifies the cells failing the tolerance requirements. The descendents of these cells at the next lower level in the hierarchy, identified by the subpointers stored in array *subp*, are temporarily loaded into the vector *asubp*. The call to WHENNE, another intrinsic CRAY subroutine, selects those descendent cells which actually contain particles, i.e., those with $subp > 0$. Finally, the cells to be examined at the next lower level are placed on the list *node*. If any further cells are to be tested, then the cycle is repeated; otherwise the tree search has been completed.

The loops 20, 30, 40, and 60 are vectorized. The CRAY intrinsic functions used are highly optimized and similar routines exist on many other machines, such as CONVEXs. For portability it is trivial to write routines to perform these CRAY functions.

Note that in the listing of *treewalk*, I have suppressed diagnostic error checks. Thus, the loops which assign values to the vectors *terms*, *asubp*, and *node* should be preceded by statements to guarantee that array overflows will not occur.

(c) *Tree Construction*

For geometric trees, such as those assumed here, the routines initializing the tree structure are also fully vectorizable. The process of setting up pointers can be vectorized by looping over all particles not yet placed in the tree (e.g., [17]). At a given level in the hierarchy the appropriate direction in the tree for each remaining particle is determined from its spatial coordinates. Those found to be the sole occupants of a cell at the next lower level are added to the tree and removed from the list of particles in the loop. The procedure is repeated until all particles are in place.

Given the pointers, the calculation of the mass, center of mass coordinates, and quadrupole moments of each node can be vectorized by looping over all cells at the same level in the tree, starting with the level immediately above the particles. Since the properties of a node depend only on the properties of its descendents, there will be no conflict if the cells being processed are at the same level and this procedure starts at the deepest level. This procedure is analogous to that given above, except that the tree traversal proceeds from the leaves to the root. The following subroutine demonstrates this concept by initializing the masses of all cells in the hierarchy.

```
      SUBROUTINE cellmass(ncell, cell)
      INCLUDE "treedefs.h"
      INTEGER asubp(maxnnode), cell(maxncell), fcell, i, isubset (maxnnode),
 &             j, lcell, ncell, nnodes, parent (maxnnode)
      DO 10 i = 1, ncell
        mass(i) = 0.0
 10   CONTINUE
      fcell = 1
 20   CONTINUE
      IF(fcell.LE.ncell) THEN
        DO 30 i = fcell, ncell
          IF(ABS(size(cell(i))-size(cell(fcell))).LT.0.01*size(cell(fcell))) THEN
            lcell = i
          ELSE
            GO TO 40
          ENDIF
 30     CONTINUE
 40     CONTINUE
        DO 70 j = 1,8
          DO 50 i = fcell, lcell
            asubp(i-fcell + 1) = subp(cell(i), j)
 50       CONTINUE
          CALL WHENIGT(lcell-fcell + 1, asubp, 1, 0, isubset, nnodes)
          DO 60 i = 1, nnodes
            parent(i) = cell(isubset(i) + fcell-1)
            asubp(i) = subp(parent(i), j)
            mass(parent(i)) = mass(parent(i)) + mass(asubp(i))
 60       CONTINUE
 70     CONTINUE
        fcell = lcell + 1
        GO TO 20
      ENDIF
      RETURN
      END
```

The argument *cell* is a list of the cells in the hierarchy, the number of which is given by the argument *ncell*. The parameter *maxnnode* has the same meaning as in *treewalk*, while *maxncell*, which is also defined in *treedefs.h*, limits the total number of cells.

For a geometric tree, it is convenient to group cells by their linear size, since it is the same for all cells at the same depth in the tree. The routine *cellmass* assumes that the cells in the list of cells are pre-ordered by their linear size. This will be the case if the tree is constructed using the procedure discussed above.

Masses are initialized in the loop 10. The variables *fcell* and *lcell* identify the first and last cells in the sorted list of cells that are the same size, *i.e.*, at the same level in the tree. The pointer *fcell* is set to 1, prior to the main block of statements beginning with the continuation line 20.

The masses of the cells in the current group are computed within the major IF–THEN block following line 20, as long as some cells remain to be processed. The statements within loop 30 determine which cells in the sorted list are at the same level as the cell identified by *fcell*. Loop 70 is over the eight possible descendants of each cell in the group. For each possible path to the next lower level, the loop 50 initializes a temporary vector, *asubp*, containing the subpointers of the current group of cells. The call to WHENIGT, another CRAY intrinsic function, selects those descendent cells which actually contain particles. The masses of those parent cells with non-empty descendents are then incremented in loop 60. Finally, after all descendents have been checked, *fcell* is set to point to the next unprocessed cell in the sorted list, and the procedure is repeated.

The loops 10, 50, and 60 are vectorized. The loop 30 is not, but requires a negligible amount of time. Again, diagnostic error checks have been suppressed in this listing.

(d) *Other Applications*

The algorithm outlined in Section IIIb can also be applied to nearest neighbor searches using trees [12]. That is, suppose the neighbors lying within a distance r_s of body p are to be located. The tree traversal proceeds as before, level by level, checking to see if this search volume overlaps the node currently being examined. If not, then this cell can be neglected; otherwise the cell is subdivided and its non-null descendents are then examined in the next iteration. This process continues until a body node is found, and then a final check is made to determine if it is, in fact, a neighbor of body p.

(e) *Timing Tests*

To ascertain the actual gain in *cpu* efficiency from vectorization of the tree traversals in the Barnes–Hut algorithm, I ran a number of models, similar to those considered earlier by Hernquist [15], varying N, θ, and the number of terms in the multipole expansions. All tests were performed on a CRAY X-MP 48, using the CFT FORTRAN compiler, version 1.14.

In Table I the results of timing tests made with equilibrium Plummer models are shown for a fully vectorized tree code and Hernquist's partially vectorized code [15]. Here $N = 8192$ and only monopole terms were retained in the multiple expansions. The corresponding results for a quadrupole version are given in Table II.

As θ decreases, the enhancement in performance from vectorization becomes

TABLE I

CPU Efficiency of Monopole Code

θ	Partial vectorization (cpu s/step)	Full vectorization (cpu s/step)	Ratio (net gain)
1.0	5.95	2.53	2.35
0.9	7.04	2.90	2.43
0.8	8.67	3.41	2.54
0.7	11.3	4.24	2.67
0.6	15.4	5.47	2.82
0.5	22.4	7.59	2.95
0.4	33.9	11.2	3.04

more significant, as the average length of the vectors being processed increases. The gain appears to saturate around a factor of ≈ 3 for the monopole version and a factor ≈ 2.5 for the quadrupole code. The improvement is larger if only monopole terms are used since the tree traversals then account for a relatively larger fraction of the *cpu* costs (e.g., [15]). For larger N, the gain increases slowly, as the vectors become longer, and appears to saturate at a level similar to that indicated in Tables I and II.

The cpu usage of the fully vectorized code is more evenly balanced between tree descents and force summation than in the partially vectorized code, which was dominated by the former. For the tests given in Table I, roughly 70% of the time was spent in performing tree searches and $\approx 25\%$ in force summation. For the quadrupole code, Table II, the percentages are nearly equal, with $\approx 45\%$ in the tree descent phase and $\approx 50\%$ in force summation. In principle, some improvement may be possible for the monopole code through the use of Barnes' technique [16] for establishing interaction lists for groups of particles.

These timing tests include a full vectorization of the subroutines used to build

TABLE II

CPU Efficiency of Quadrupole Code

θ	Partial vectorization (cpu s/step)	Full vectorization (cpu s/step)	Ratio (net gain)
1.0	8.98	4.19	2.14
0.9	10.5	4.78	2.20
0.8	12.9	5.61	2.30
0.7	16.7	7.14	2.34
0.6	22.9	9.09	2.52
0.5	32.3	12.9	2.50
0.4	47.7	18.2	2.62

and manipulate the tree structure. These procedures are inexpensive, in spite of the fact that the tree is reconstructed at each step. For the fully vectorized runs shown in Tables I and II, the overhead in building the tree amounted to ≈ 2–3% of the cpu time used, an improvement over the partially vectorized code [15]. Although not critical in this case, it is important that the tree construction be performed as efficiently as possible for applications where particles have individual time steps and the tree is rebuilt on the smallest time-scale [12].

Finally, the efficiency of the fully vectorized code, running on a CRAY X-MP48, was compared with an identical version operating on a SUN 3/50 and a VAX 11/780. The performance ratio CRAY:SUN or CRAY:VAX was typically ≈ 350–500, depending on the vector lengths, which, in turn, depend on N and θ. For example, if $\theta = 0.6$, with quadrupole terms, an $N = 8192$ Plummer model ran a factor ≈ 440 times faster on the CRAY than the SUN, in good agreement with tests made with linear algebra packages (e.g., [22]). The SUN 3/50 has a floating point rating of roughly 90 kflops, implying that the CRAY X-MP48 is being driven at ≈ 35–50 Mflops, which is still considerably less than the theoretical peak performance of 210 Mflops per processor. This is probably the result of memory access delays, lack of complete chaining, operations involving vectors of non-optimal length, and extensive use of gather–scatter operations.

IV. Summary

A general technique for vectorizing tree traversals has been discussed. In this approach, trees are no longer regarded as a collection of individual nodes, but rather as a set of vectors of variable width. Operations on these vectors can be fully vectorized, provided that indirect addressing does not inhibit vectorization. Simple tests indicate that factors ~ 4–5 improvement in the efficiency of tree traversals can be obtained with this technique on machines such as CRAY X-MPs.

Contrary to earlier somewhat pessimistic expectations, the tree traversals required by the hierarchical tree method can be vectorized in this manner, resulting in *overall* factors ~ 2–3 reduction in cpu costs.

It is perhaps somewhat surprising that tree traverals can be fully vectorized since such functions would appear to be most ideally suited for parallel architectures. Yet, the prescription presented here is not even unique. It appears that the algorithms proposed thus far will complement one another, implying that the hierarchical tree method can be adapted to perform efficiently on almost any computer. The technique outlined in this paper will be optimal on machines with hardware gather–scatter which prefer short vectors, such as CRAYs, while Makino's method, everything else being equal, will be most effective on machines which prefer longer vectors. Finally, as noted by Barnes [16], the limitations imposed by the requirement of a hardware gather–scatter capability can be minimized by shifting the emphasis to linear arrays. The primary advantages of the approach presented here lie in its simplicity and potential for wide-spread application.

ACKNOWLEDGMENTS

I thank Josh Barnes, Piet Hut, and Jun Makino for discussion about the hierarchical tree method. This work was supported in part by a grant from the Pittsburgh Supercomputing Center and by New Jersey High Technology Grant 88-240090-2.

REFERENCES

1. A. W. APPEL, Thesis, Princeton University, 1981 (unpublished).
2. A. W. APPEL, *SIAM J. Sci. Statist. Comput.* **6**, 85 (1985).
3. J. G. JERNIGAN, in *IAU Symp. 127 Proceedings. Dynamics of Star Clusters*, edited by J. Goodman and P. Hut (Reidel, Dordrecht, 1985), p. 275.
4. D. PORTER, Thesis, University of California, Berkeley, 1985 (unpublished).
5. J. BARNES AND P. HUT, *Nature* **324**, 446 (1986).
6. L. GREENGARD AND V. ROKHLIN, *J. Comput. Phys.* **73**, 325 (1987).
7. L. GREENGARD, *The Rapid Evaluation of Potential Fields in Particle Systems* (MIT Press, Cambridge, 1988).
8. J. AMBROSIANO, L. GREENGARD, AND V. ROKHLIN, *Comput. Phys. Commun.* **48**, 117 (1988).
9. J. J. MONAGHAN, *Comput. Phys. Commun.* **48**, 89 (1988).
10. W. BENZ, *Comput. Phys. Commun.* **48**, 97 (1988).
11. L. HERNQUIST, *Comput. Phys. Commun.* **48**, 107 (1988).
12. L. HERNQUIST AND N. KATZ, *Ap. J. Suppl.* **70**, 419 (1989).
13. D. H. BOAL, *Ann. Rev. Nucl. and Part. Sci.* **37**, 1 (1987).
14. W. D. HILLIS, *The Connection Machine* (MIT Press, Cambridge, 1985).
15. L. HERNQUIST, *Ap. J. Suppl.* **64**, 715 (1987).
16. J. BARNES, *J. Comput. Phys.* **87**, 161 (1990).
17. J. MAKINO, *J. Comput. Phys.* **87**, 148 (1990).
18. J. CARRIER, L. GREENGARD, AND V. ROKHLIN, *SIAM J. Sci. Statist. Comput.* **9**, 669 (1988).
19. J. BARNES, in *The Use of Supercomputers in Stellar Dynamics*, edited by P. Hut and S. McMillan (Springer-Verlag, Berlin, 1986), p. 175.
20. D. PORTER AND J. G. JERNIGAN, preprint (1987).
21. W. H. PRESS, in *The Use of Supercomputers in Stellar Dynamics*, edited by P. Hut and S. McMillan (Springer-Verlag, Berlin, 1986), p. 184.
22. J. J. DONGARRA, Technical Memorandum No. 23, Argonne National Laboratory, 1988 (unpublished).