JOURNAL OF COMPUTATIONAL PHYSICS 87, 148–160 (1990)

Vectorization of a Treecode

JUNICHIRO MAKINO*

Institute for Advanced Study, Princeton, New Jersey 08540

Received July 19, 1988; revised January 24, 1989

Vectorized algorithms for the force calculation and tree construction in the Barnes–Hut tree algorithm are described. The basic idea for the vectorization of the force calculation is to vectorize the tree traversal across particles, so that all particles in the system traverse the tree simultaneously. The tree construction algorithm also makes use of the fact that particles can be treated in parallel. Thus these algorithms take advantage of the internal parallelism in the N-body system and the tree algorithm most effectively. As a natural result, these algorithms can be used on a wide range of vector/parallel architectures, including current supercomputers and highly parallel architectures such as the Connection Machine. The vectorized code runs about five times faster than the non-vector code on a Cyber 205 for an N-body system with $N = 8192$. © 1990 Academic Press, Inc.

1. INTRODUCTION

The most time-consuming part of an N-body code is the force calculation. A naive approach requires $O(N^2)$ computing cost. Recently, several "tree" algorithms have been developed which map a hierarchical tree structure on an N-body system [1–4]. In these algorithms, the force from distant particles is replaced by the force from their center of the mass. Tree structures provide a systematic way of performing this replacement, thereby reducing the cost of the force calculation from $O(N^2)$ to $O(N \log N)$.

Another algorithm which has an even better asymptotic behavior of the computational cost scaling as $O(N)$ has been proposed [5–7]. However, all currently existing implementations of this $O(N)$ algorithm in three dimensions are limited to the uniform tree, in which the physical space is divided into cells of a constant size. This uniform tree is not suitable for astronomical applications where the spatial inhomogeneity tends to be very large. Thus we do not discuss the $O(N)$ algorithm here.

Practically, the difference in the actual computational cost between the $O(N \log N)$ schemes and $O(N)$ schemes is not likely to be large for $N \lesssim 10^6$. Katzenelson [8] discussed that the $O(N \log N)$ scheme requires six times more

* Present address: Department of Earth Sciences and Astronomy, College of Arts and Sciences, University of Tokyo, Tokyo, Japan.

computing time than the $O(N)$ scheme for 1250k particles on the Connection Machine CM-2. However, in his estimate the amount of computation of $O(N \log N)$ scheme is overestimated roughly by a factor of 6. In [8], it is assumed that

$$n_{\text{cells}} = 189(l-2), \tag{1}$$

where n_{cells} is the number of cells that interact with a particle and l is the level of the tree. This number is correct for the implementation of the $O(N \log N)$ scheme discussed in [8]. For the usual implementation (e.g., [4]), however, Makino [9] showed that

$$n_{\text{cells}} \sim 30(l-2), \tag{2}$$

for the accuracy comparable to that obtained by the most accurate $O(N)$ calculation discussed in [8].

At present, the fastest computers available are vector machines with pipeline architectures. The recursive structure of the algorithm makes it difficult to implement the tree algorithm effectively on vector machines. The standard Fortran 77 does not allow recursive subroutines. We must unroll the recursive structure of the algorithm into a sequence of iterative operations. Several compilers such as the Cray CFT accept recursive subroutines. However, the performance of the recursive algorithm on vector machines tends to be rather poor, because it does not make use of vector pipelines. The iterative algorithm exhibits similar low performance, because the nature of the iteration prohibits the vectorization. Thus, we cannot take advantage of the superior performance of the vector machines unless we find some way to reformulate the force calculation to adapt to the vector machines.

It is clear that forces on distinct particles can be evaluated independently. This is the most fundamental parallelism in an N-body calculations. Thus, we can let all particles of the system traverse the tree simultaneously. Barnes [10] described an implementation of this parallel force evaluation on a fine-grained parallel machine. His implementation, however, was strongly affected by both hardware and software limitations. Thus it was rather complicated and not very efficient. Here we give a simple algorithm to realize the parallel tree traversal. This algorithm has been implemented on vector machines with no fundamental difficulty. Moreover, it has been implemented effectively on highly parallel machines [11] or coarse-grained parallel machines.

After the force calculation is effectively vectorized, the cost of the tree construction remains to be improved. In Section 3, we discuss a vectorizable algorithm for the construction of the octtree used in the Barnes–Hut algorithm. Barnes [10] described a parallel tree construction based on a bottom-up scheme. His algorithm used the bit operations intensively. Unfortunately, bit operations are not very efficient on the vector processors which are designed to perform floating point operations effectively. Here, we describe a simple and intuitive algorithm of top-down tree construction that can be effectively vectorized. The calculation of the mass and the center of the mass coordinates of the cells is also vectorizable as discussed by Hernquist [12]. Thus, all aspects of the tree algorithm are effectively vectorizable.

In Section 4 we give the result of timing benchmarks. Our vectorized code runs about five times faster than the original code and can outperform the direct summation code for $N \gtrsim 10^3$ with average errors in the force of $\sim 1\%$ on a Cyber 205. We will also give the result of timing runs on other supercomputers; Cray X-MP/1 and two Japanese supercomputers. In Section 5 we briefly discuss the implementation of the present algorithm on parallel machines, especially coarse-grained parallel machines like Cray X-MP/4 or ETA-10.

2. Vectorized Tree Traversal

In the original Barnes–Hut algorithm, the force calculation is described as a recursion:

Algorithm (a). Recursive tree-force calculation for particle *i*.

```
subroutine treeforce(i, node)
  if (node and particle i are well separated)
    force = force from the total mass in the center of mass of node
  else
    force = sum of forces from the children of node
  endif
  return
```

This algorithm is recursive, because the **else** block implies the calling of the subroutine **treeforce** itself. We use the following condition to determine if a node is well separated from a particle:

$$l/d < \theta, \tag{3}$$

where l is the size of the cube corresponding to the node, d is the distance between the cube and the particle, and θ is the parameter which determines the accuracy and thereby the amount of computation needed. Figure 1 shows how this algorithm works. To obtain the gravitational force on the particle indicated by X, we start at the root of the tree. The root is usually not well separated from the particle. Therefore, we descend the tree and try to evaluate the total force as a sum of the forces from the children of the root. If a child-node is well separated, its contribution to the force is evaluated. If not, we further descend the tree recursively until we reach nodes which are well separated or leaves which contain single particles (which are by definition well separated).

In a usual sequential language such as C or Pascal, the above algorithm can be implemented directly using recursive functions. The actual execution of the above algorithm in these sequential languages results in a depth-first traversal of the tree.

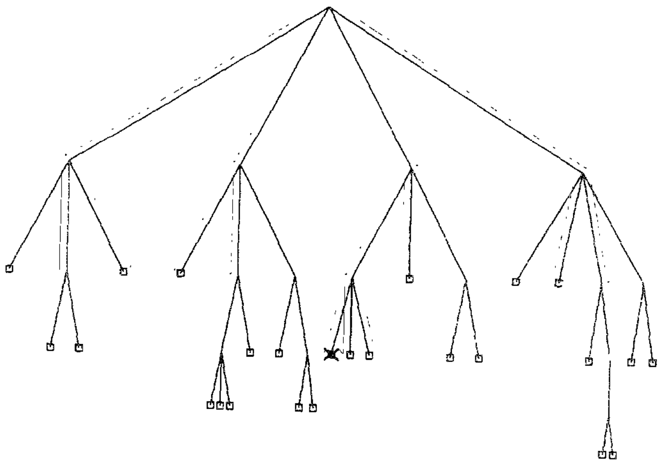

FIG. 1. Recursive tree traversal. X indicates the particle on which the force is calculated. Arrows indicate how the calculation goes on.

In order to implement this depth-first search using Fortran 77, we must unroll the recursive procedure into an iterative procedure. Such a formulation was first introduced by Barnes [13] who used stacks explicitly to transform the recursion into iteration. This algorithm is vectorizable across partricles. In the vectorized code, every particle has its own stack. Thus, we have a vector of stacks. In each iteration, we apply vector operations on this vector of stacks. Note that the content of each stack differs for different particles. Though the same operation is applied on all particles, the result is not the same because particles have different spatial coordinates.

On a scalar machine, this stack-based algorithm is simple and runs efficiently. Nevertheless, on vector machines, the stack manipulation requires complicated coding resulting in considerable performance degradation. Here, we describe an algorithm which does not require stack manipulation. The resulting code is simpler than the stack-based one and runs faster.

In our method, the depth-first search is reformulated into a purely iterative form. Then, we vectorize that algorithm across particles. In order to perform the depth-first search we do not require the stack, if we have "preprocessed" the tree so that we can obtain the necessary information.

The basic idea is to follow the "left wall" of the tree systematically. Using this trick, we can traverse the whole tree in depth-first order. In the force calculation using the tree, we can apply a similar procedure, except that we sometimes stop before we reach the leaf. Figure 2 shows how we traverse the tree. This traversal gives exactly the same result as that of the recursive traversal of Fig. 1. This procedure is schematically expressed as follows:

ALGORITHM (b). Force calculation through a stack-less iterative tree walk.

```
subroutine treeforce(i)
  node = root_node
  while (node .ne. null)
    if(node and the particle are well separated)
      force = force + (force from the center of mass of node)
      node = next(node)
    else
      node = first_child(node)
    endif
  endwhile
  return
```

Here we assume that the children of a node form an ordered set. **first_child** returns the first element of the set of children of a node. This **first_child** corresponds to the leftmost children of the nodes in Fig. 2. The jumps to right or upper-right directions in Fig. 2 are performed using the **next** function. It can return a brother-node, or an uncle-node, or a brother-of-grandfather-node, etc., depending on the position of the node in the tree. To be precise, **next** is defined as follows:

```
function next(node)
  if (node is not the last member of the children list of its parent)
    next = the next member of the parent's children list
  else
    next = next(the parent of node)
  endif
  return
```

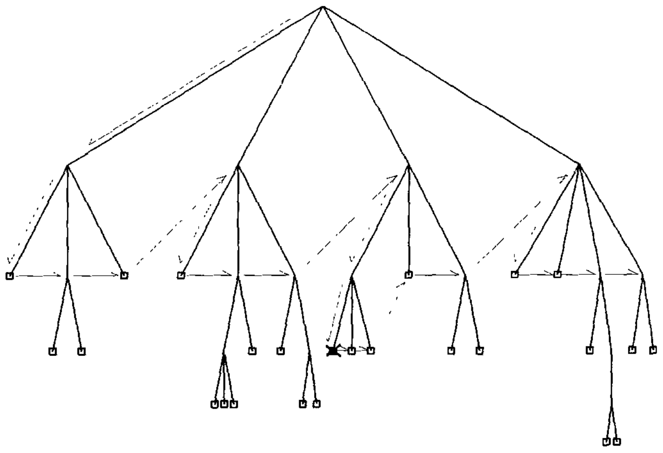

FIG. 2. Stackless tree traversal. X indicates the particle on which the force is calculated. Arrows indicate how the calculation goes on.

In the actual implementation, it is not quite efficient to call the function **next** each time. Instead we evaluate and store the **next** node for each node before we start the force calculation. The **first_child** can also be stored in similar way. Therefore, in the actual implementation, both procedures are replaced by a reference to an element of an array.

The vectorization of Algorithm (b) should be straightforward. If we had a reasonably smart compiler like C* [14] of the Connection Machine [15] on vector supercomputers, we could say:

```
do i = 1, n in parallel
   call treeforce(i)
enddo
```

Unfortunately, current compilers cannot vectorize **DO** loops containing subroutine calls. Therefore we should expand the body of the subroutine **treeforce** inline. Several compilers such as Fujituu's and the ETA Fortran have a limited ability to expand subroutines inline. This ability, however, does not help us. After the subroutine **treeforce** is expanded inline, the above loop becomes as follows:

ALGORITHM (c). Subroutine call expanded inline.

```
do i = 1, n in parallel
   while(not finished)
      do something
   endwhile
enddo
```

Here, *do something* represents the body of the subroutine **treeforce**. Due to syntax limitations in standard Fortran 77, the **while** ... **endwhile** structure needs to be implemented as a combination of an **IF** statement and a **GOTO** statement. "Vectorizing" compilers cannot vectorize **DO** loops containing a backward **GOTO**. Nevertheless, we can vectorize this by introducing a further modification:

ALGORITHM (d). Vectorized force calculation.

```
while(any one particle is not finished)
   do i = 1, n in parallel
      if(not finished)
         do something
      endif
   enddo
endwhile
```

This form is "simple" enough to be recognized as vectorizable by at least some compilers. It is somewhat surprising that no vectorizing compiler recognizes the structure of Algorithm (c) as vectorizable. A smart compiler should be able to convert Algorithm (c) to Algorithm (d) automatically. In practice, some compilers still complain that they cannot vectorize a **DO** loop which contains an **IF** statement. At this point, it is a simple, though tedious, task to rewirte the code with special vector functions or array extensions.

A significant inefficiency of Algorithm (d) is that the **DO** loop alway loops over all particles, no matter how many particles actually need to be treated. When executed in scalar mode, this fact does not cause a serious problem, because the **IF** block which is not executed requires the time to evaluate the condition only. However, when executed in vector mode, the **IF** block requires a time proportional to the loop length, irrespective of the number of iterations in which the **IF** statement is evaluated as true. This consideration leads to the following coding:

ALGORITHM (e). Vectorization of **IF** using list vector.

```
nlist = 0
do i = 1, n in parallel
  list(i) = i
enddo
nlist = n
while (nlist .gt. 0)
  do i = 1, nlist in parallel
    do something for list(i)
  enddo
  new_nlist = 0
  do i = 1, nlist
    if (particle list(i) is not finished)
      new_nlist = new_nlist + 1
      list(new_nlist) = list(i)
    endif
  enddo
  nlist = new_nlist
endwhile
```

The key idea is to use the list vector, which is a vector of indices, to keep the list of the particles which have not finished their force calculation. The body of the force calculation is applied only to those particles which appear in the list. After each iteration, the list is reconstructed to purge the particles that have finished their force calculation. Note that the last **DO** loop is vectorizable. Even if the compiler does not vectorize it, manufacturers supply vector functions which can be used in place of the **DO** loop on all machines we tested. This operation is called a "compress" operation.

We should remark that there still remains some room for improvement. In the main loop of Algorithm (e) (the *do something* loop), everything is accessed indirectly via the list vector. If we keep the "compressed" list of positions of particles, we can reduce the amount of indirect accessing. This compressed list of positions needs to be reconstructed after each iteration, in the same way as the list vector itself. Nevertheless this modification offers some improvement in speed, because on any of vector machines the "compress" operation is much faster than the indirect addressing. The improvement of speed of this final form over Algorithm (d) is in the range of 15 to 50%, depending on the compilers and the hardware. It should be noted that there is at least one compiler which automatically generates the list vector for an **IF** statement and thus makes the modification from Algorithm (d) to (e) unnecessary (Fujituu Fortran 77/VP).

3. Vectorized Tree Construction

The most intuitive way to construct an octtree for a three-dimensional N-body system is the following: First, we make a box large enough to contain the whole system. Then we repeatedly subdivide the boxes into eight subboxes until each box at the leaf of the tree contains only one particle. In practice, there are two possible algorithms to implement the procedure described above. The first one is the depth-first search. In this method, the subbox first created is immediately subdivided. This process continues until we reach the leaf. Then we go up this tree, until we reach the box which has a subbox that is not fully subdivided. Then we complete this subbox in a similar way and repeat this process until we complete the whole tree. This method is directly implemented by recursive procedures. However, there is little parallelism.

The second one is the breadth-first search. In this method, all subboxes in the same level of the tree are subdivided simultenously. Implementation of breadth-first search requires considerably more bookkeeping than simple recursive formulation of the depth-first search. Nevertheless, by using the breadth-first search, we can obtain a much higher degree of parallelism, because all boxes in one level can be treated in vector mode. The algorithm is schematically described:

Algorithm (f). Vectorized tree construction.

```
create the root node
attach all particles to the root
set flags for all particles as not_finished
$l = 1$
while(for any i, flag(i) = not_finished)
  for each particle with the flag not_finished
    determine which subbox it is contained
  endfor
  for all subbox in level $l$
    count the number of particles in it
  endfor
```

```
    for all subbox in level $l$
      if (it contains at least one particle)
        create the corresponding node, connect it to its parent
      endif
    endfor
    for each particle with the flag not_finished
      if (the cell which it attached to contain only one particle)
        setthe flag as finished
      endif
    endfor
    $l = l + 1$
endwhile
```

With the above algorithm, we first create the root node, which is large enough to contain all particles. Then we construct the tree level by level. At the first step, we determine which subnode of the root node each particle is contained in. If any subnode contains no particles, we do not create that node. If any subnode contain only one particle, that particle is labeled as **finished**. Then we repeat the above process for the subnodes of the root node that contain more than one particles. We repeat this process until all particles are labeled as **finished**. The actual implementation is quite a bit more complicated than the schematic form of Algorithm (f), because of difficulties similar to those we encountered in Section 2.

The calculation of the mass and center of mass coordinates is also vectorizable by treating all nodes in the same level of the tree simultaneously as discussed in [12].

4. Timing Results

For calculations with $N \gtrsim 10^3$, our vectorized code runs about five times faster than the partially vectorized code of Hernquist [16] on a Cyber 205. We can evaluate the force on all particles in a 1024-body system in 0.3 s or an 8192-body system in 3.2 s within an accuracy $\sim 1\%$ ($\theta = 1$, Plummer model). If we compare this with Hernquist's [12] fully vectorized code on a Cray X-MP, our code on a Cyber is slightly slower. Hernquist [12] gives a timing of 2.5 s/step for an 8192-body system on a Cray X-MP. Our algorithm runs somewhat faster on a Cray X-MP than on a Cyber, thus resulting in a speed comparable to that of Hernquist's [12] code.

Table I gives the timing for one integration step for an 8192-body Plummer model on several supercomputers with $\theta = 1.0$. All timings are performed using the full precision (64 bit) arithmetic and the monopole moment only. It should be noted that our code is specially optimized for the Cyber and the timing on other machines may not indicate their best performance. The highest absolute speed is attained by a Hitac S-820, which is about five times faster than a Cray or a Cyber.

TABLE I

Performance of the Vectorized Code

Machine		Treecode (cpu s/step)	Direct Summation (cpu s/step)	Net gain by treecode
Cyber 205[a]	(FTN200)	3.24	16.0	4.9
Cray X-MP/18	(CFT77)	2.83	11.6	4.1
Facom VP-400	(FORTRAN77/VP V10L20)	1.79	2.93	1.6
Hitac S-820/80	(FORT77/HAP V21-0A)	0.67	1.73	2.6

[a] 2 pipes.

However, the relative speedup from the $O(N^2)$ direct summation is somewhat lower for Japanese machines.

The efficiency of our method over the scalar version does not depend on the value of θ. Other methods [12, 17] tend to show higher efficiency for smaller value of θ. The reason for this difference is quite simple. Our method vectorizes the calculation across particles. Thus the vector length is always equal to N. Other schemes vectorize the force calculation for one particle. Thus for smaller θ they get longer vectors and therefore better efficiency.

Table II shows the cpu time per step on a Cyber. The number of particles and other parameters are the same as in Table I, unless specified otherwise. The term half-precision stands for the 32 bit arithmetic. The half-precision version is about 20% faster than the full-precision version. The theoretical peak speed of a Cyber for the half-precision calculation is twice that for the full-precision calculation. The reason why the obtained speedup is much smaller than the theoretical factor of two is that the bottleneck of our algorithm is the indirect memory accessing, for which the speed is same for both full- and half-precision calculations.

The quadrupole version is about 70% slower than the monopole version both for the half- and full-precision calculations. This factor depends on θ very weakly and is similar to that Hernquist [12] obtained on a Cray X-MP. On Japanese machines

TABLE II

Performance on Cyber

		cpu time per step (s)
Monopole	Half precision	2.60
Quadrupole	Half precision	4.13
Monopole	Full precision	3.24
Quadrupole	Full precision	5.37

the increase of CPU time by using quadrupole moments is less than 30%. Japanese machines exhibit less speed down than a Cyber simply because the efficiency of the monopole version is lower on Japanese machines than on a Cyber.

5. Discussion

5.1. *Parallel Machines*

The algorithms described in Sections 2 and 3 have been successfully implemented on a highly parallel machine, the Connection Machine [15]. The detail of this implementation is described elsewhere [11]. Unfortunately, the relative efficiency compared to an $O(N^2)$ code is even lower than the lowest value obtained on vector machines. The absolute performance is, however, similar to that of a Cyber or a Cray.

We have not tried to implement our algorithm on MIMD multiprocessor supercomputers like Cray X-MP/4 or ETA-10 yet. The implementation, however, seems to be quite straightforward. With the tree algorithm, the most expensive part is the force calculation using the tree. The cost of the tree construction is much smaller, typically a few percent of the total cost. Therefore we can safely limit our attention to the force calculation only. The tree construction will be performed on one processor. The task of force calculation will be distributed to all available processors. If we have n processors, each will hold N/n particles. Then, the force on those particles is evaluated using Algorithm (d). The details of the implementation will be largely machine dependent. For example, on a machine with a shared-memory architecture like the Alliant FX, we can actually use Algorithm (d) without any modification. The compiler of the Alliant FX is smart enough to produce a machine code that automatically distributes the task to all available processors. On machines that have large and fast local memory, the most effective procedure would be to keep copies of the whole tree on the local memory of each processor.

On an Alliant FX-8 with four processors, we obtained a speedup of factor 3.5 using four processors. This seems quite satisfactory for what we can expect from the automatic parallelization. On other machines we expect similar performance.

5.2. *Conclusion*

We have discussed vectorizable algorithms for the force calculation using trees and for the tree construction itself. Both algorithms are vectorized across particles. Therefore the degree of parallelism, i.e., the vector length, is equal to the particle number N. This long vector length is the primary advantage of our algorithm, because it promises reasonable performance on any vector hardware, provided it has the ability to vectorize indirect addressing (gather/scatter function). We obtained a performance similar to that of Hernquist's [12] vectorized code on a Cray X-MP. On a Cyber perhaps our code would show better performance than Hernquist's code, because the vector length is much longer for our code and because a Cyber requires a much longer vector length than a Cray for good performance. Thus, our code runs effectively on a wide range of vector machines. This implies that our code is "portable," at least on vector processors. For Japanese machines we can use the same code written in standard Fortran 77. If a full implementation of the CFT77 compiler was available, we could also use a Fortran 77 version on a Cray.

Our algorithm is applicable to both fine-grained and coarse-grained parallel machines. With a smart compiler, the Fortran 77 version of our code runs efficiently on a coarse-grained parallel machine, as was shown in the case of Alliant FX-8. Thus we can conclude that the tree algorithm can be effectively used on any type of fast machine that is currently available or will be available in the near future.

There is one flaw in our method, however. The fact that we vectorized the force calculation across particles makes it very difficult to use individual time steps. Hernquist's [12] method is better suited for this purpose. However, by applying the block timestep method (e.g., [18]), we can still obtain some gain from our algorithm. Finally, we should note that our method and Hernquist's method can be applied simultaneously. Our method and his method vectorize the force calculation in different directions. By careful coding we can combine these two methods.

Acknowledgments

I thank Joshua Barnes and Lars Hernquist for providing their code and for useful discussions. I thank Piet Hut for stimulating discussions. Computations were performed at John von Neumann National Supercomputer Center, Princeton, Computer Center of University of Tokyo, Japan, and Institute for Supercomputing Research, Tokyo, Japan.

References

1. A. Appel, *SIAM J. Sci. Statist. Comput.* **6**, 85 (1985).
2. J. G. Jernigan, "Direct *N*-Body Simulations with a Recursive Center of Mass Reduction and Regularization," in *Dynamics of Star Clusters, I.A.U. Symp. 113*, edited by J. Goodman and P. Hut (Dordrecht, Reidel, 1985), p. 275.
3. D. Porter, Ph D. thesis, Physics Department, University of California, Berkeley, 1985 (unpublished).
4. J. Barnes and P. Hut, *Nature* **324**, 446 (1986).
5. L. Greengard and V. Rokhlin, *J. Comput. Phys.* **73**, 325 (1987).
6. L. Greengard, *The Rapid Evaluation of Potential Fields in Particle Systems* (MIT Press, Cambridge, MA, 1988).
7. F. Zhao, Master's thesis, Department of Electrical Engineering and Computer Science, MIT 1987 (unpublished).
8. J. Katzenelson, *SIAM J. Sci. Statist. Comput.*, in press.
9. J. Makino, *J. Comput. Phys.*, in press.
10. J. Barnes, "An Efficient N-Body Algorithm for a Fine-Grain Parallel Computer," in *The Use of Supercomputers in Stellar Dynamics*, edited by S. L. W. McMillan and P. Hut, (Springer-Verlag, Dordrecht, 1986), p. 175.
11. J. Makino and P. Hut, *Comput. Phys. Rep.* **9**, 199 (1989).

12. L. HERNQUIST. *J. Comput. Phys.* **87**. 137 (1990).
13. J. BARNES, unpublished preprint, 1987.
14. J. ROSE AND G. L. STEELE, "C*: An Extended C Language for Data Parallel Programming," Thinking Machines Corporation Technical Report PL87-5, 1987 (unpublished).
15. W. D. HILLIS, *The Connection Machine* (MIT Press, Cambridge, MA, 1985).
16. L. HERNQUIST, *Ap. J. Suppl.* **76**, 64 (1987).
17. J. BARNES, *J. Comput. Phys.* **87**, 161 (1990).
18. S. L. W. MCMILLAN, "The Vectorization of Small-N Integrators" in *The Use of Supercomputers in Stellar Dynamics*, edited by S. L. W. McMillan and P. Hut (Springer-Verlag, Dordrecht, 1986), p. 156.