View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

NASA Contractor Report 199882

An N-body Tree Algorithm for the Cray T3D

Kevin M. Olson and Charles V. Packer

MAY 1996

NASA Contractor Report 199882

An N-body Tree Algorithm for the Cray T3D

Kevin M. Olson
George Mason University
University Drive,
Fairfax, Virginia 22030

Charles V. Packer
Hughes STX Corporation
4400 Forbes Blvd.
Lanham, Maryland 20706

Prepared for
Goddard Space Flight Center
under Grant NAG5-2652

National Aeronautics and
Space Administration

Scientific and Technical
Information Program

This publication is available from the NASA Center for AeroSpace Information,
800 Elkridge Landing Road, Linthicum Heights, MD 21090-2934, (301) 621-0390.

Abstract

We describe in this paper an algorithm for solving the gravitational N-body problem using tree data structures on the Cray T3D parallel supercomputer. This implementation is an adaptation of previous work where this problem was solved using a SIMD, fine-grained parallel computer. We show here that this approach lends itself, with small modifications, to more coarse-grained parallelism as well. We also show that the performance of the algorithm on the Cray T3D parallel architecture scales adequately with the number of processors (up to 256). Specific changes to the basic algorithm are also described which allow greater performance levels to be reached using the Cray T3D parallel architecture. A peak performance level of 9.6 Gflop/s is reached on 256 processors for the time critical gravity computation.

1 Introduction

For many problems in astrophysics the first order effect determining the dynamical evolution of the system is the force of gravity. Further, a large portion of such systems can be described as a system of gravitationally interacting masses. Some examples are star clusters, galaxies, clusters of galaxies, and the large scale structure of the universe.

The gravitational N-body problem is defined by the following simple relation. The force on particle i in a system of N gravitationally interacting particles is given by,

$$\vec{F_i} = \sum_{j=1}^{N} \frac{-Gm_i m_j \vec{r_{ij}}}{(r_{ij}^2 + \epsilon^2)^{3/2}} \tag{1}$$

where G is the universal gravitational constant, m_i and m_j are the masses of particles i and j, $\vec{r_{ij}}$ is the position vector separating them, and ϵ is a smoothing length which can be nonzero and serves to eliminate diverging values in $\vec{F_i}$ when $\vec{r_{ij}}$ is small. This parameter also serves to define a resolution limit to the problem. The negative sign indicates that the force is attractive. This equation also shows that the problem scales as N^2. Once the forces above are computed, the particles positions are advanced in time by integrating Newton's equations of motion (an $O(N)$ process). Since the force computation given by the above equation scales as N^2 the size of a simulation is restricted to several thousand particles. This is orders of magnitude lower than the real number of particles in the systems mentioned above and does not even give the dynamic range that we would like to achieve to answer some of the most basic scientific questions raised by observations of these systems. Therefore, we would like to increase the particle number in a typical simulation to be as large as possible. To do this, approximate techniques have been developed, one of which employs tree data structures.

Tree codes are a collection of algorithms which approximate the solution to Eq. 1 [1,2,8]. In these algorithms the particles are sorted into a spatial hierarchy which forms a tree data structure. Each node in the tree then represents a grouping of particles. Data which represents average quantities of these particles (e.g. total mass, center of mass, and high order moments of the mass distribution) are computed and stored at the nodes of the tree. The forces are then computed by having each particle search the tree and pruning subtrees from the search when the average data stored at that node can be used to compute a force on the searching particle below a user supplied accuracy limit. For a fixed level of accuracy this algorithm scales as $Nlog(N)$

although $O(N)$ algorithms are also possible.

Since the tree search for any one particle is not known *a priori* and the tree is unstructured, frequent use is made of indirect addressing. This presents problems for distributed memory, parallel implementations of this algorithm since one wishes to minimize any off processor accesses of data. On the other hand, the problem does possess a highly parallel component: each particle searches the tree structure completely independently of all other particles in the system. This fact has been exploited by several groups to develop implementations of this algorithm for vector computers [7] and also coarse-grained parallel computers [4,11]. Olson and Dorband [10] have also implemented a tree algorithm for the solution of this problem on a SIMD, fine-grained parallel architecture. In this implementation a balanced binary tree structure (the number of children of each parent node in the tree was two) was used to facilitate the data layout on the processor array of the Maspar MP-2 on which they implemented this algorithm. The use of a regular, balanced tree also allows the computation of neighboring nodes in the tree as opposed to following pointers. We discuss in the next two sections how this algorithm was modified for a coarse-grained, message passing architecture.

2 The Balanced Tree Algorithm

We briefly discuss here the balanced tree algorithm described by Olson and Dorband [10]. The tree is constructed using an algorithm termed 'recursively bisect the longest dimension' (RBLD). In this algorithm the dimension (x, y, or z) which spans the largest spatial range for all the particles distributed in space is determined. The particle data are then sorted on this dimension and the list is divided into two equal halves. This results in two sublists. This process is then applied to each of these sublists independently of the other list resulting in four independent lists. This process is then repeated recursively until each sublist contains the data of only one particle. As a result of applying this algorithm, particles which are nearby spatially are also nearby in the sorted list. To construct the first level of parent nodes in the tree, neighboring particles in the sorted list are paired and their total mass and center of mass are computed. A size is also associated with each node by finding the particle with the largest distance from the just-computed center of mass. The remaining levels in the tree are then constructed in a similar fashion by using the data stored at the previous level in the tree. Although the original implementation of this basic algorithm employed a binary tree

structure it has since been found that using an oct tree (8 children per parent node) results in roughly a factor of 2 speed up in searching the tree. Such a tree is constructed in the same fashion as the binary tree except that parent nodes in the tree are constructed from 8 nodes at the previous level. The RBLD procedure guarantees that groupings of 8 particles in the list of sorted particles also represent a small locality in space. Since the tree is still balanced, this would require using only powers of 8 numbers of particles. We overcome this limitation by simply allowing the first level below the root node to have a varying number of nodes so that powers of 2 numbers of particles can be accommodated.

Once such a tree is constructed the search is straightforward and differs little from other search strategies. A tree node is accepted for the computation of a force if,

$$r > \frac{2S}{\theta}, \tag{2}$$

where r is the distance to the node, S is its size as defined above, and θ is a user supplied parameter which allows the accuracy of the calculation to be varied. If a node in the tree is accepted then a force is computed using the data stored at that node and the subtree below that node is pruned from the search. Otherwise, the node is 'opened' and the children of that node are tested and either accepted or opened as needed. It is also possible to pipeline the force computations by storing the accepted node data into a long vector known as an *interaction list.* The force computations for a searching particle are then only performed at the end of its search through the tree [7]. This approach is advantageous for both vector and cache based architectures.

In the case of implementing this algorithm on the Maspar MP-2, Olson and Dorband [10] begin the tree search at the leaves of the tree so that data accesses are more evenly spread throughout the processor array. Further, they make extra copies of the upper levels in the tree so that data collisions are minimized. They also compute the address of the next node to search in the tree rather than following pointers which eliminates several indirect data accesses. On single processor implementations one may wish to simply precompute these addresses and store them as a list of indirect addresses which can be accessed as the tree is searched. This is possible since the logical structure of the balanced tree remains static (although the data at the nodes continually change). We describe below our method of implementing this algorithm on a message passing, coarse-grained parallel architecture.

3 Parallel Implementation

The first thing we concern ourselves with is the construction of the tree structure. The first step in this procedure is to perform a domain decomposition and distribute the particle data to different processors so that a processor controls particles within a local subspace of the entire domain. To do this we simply apply the RBLD algorithm to the distributed list of particles i.e., the maximum spatial dimension is found in the list, the list is sorted and then split resulting in two sublists. This algorithm is then applied again to each sublist independently of the other. This is done recursively until the sublists are all contained on single processors. In our implementation the sublists on each processor are of equal length. The sort used is a bitonic/merge sort (see [5] and references therein) This algorithm is also similar to that described by Salmon [12].

After the domain decomposition phase each processor holds an equal number of particles. At this point a tree is constructed on each processor in parallel using its local list of particles. For our purposes, these subtrees are constructed using the RBLD algorithm as described above. However, we note that any single-processor tree-building scheme could be used (e.g. Barnes-Hut) at this stage. Using the balanced trees that result from applying the RBLD algorithm has the advantage that the subtrees need not be constructed at each time step of a simulation while only the node data of the subtrees needs to be updated. At this point the tree build is complete and a 'forest' of subtrees (one per processor) results. Each tree has a logical structure which is logically identical to all other subtrees and each represents a local region of space. In the implementation discussed here the center of mass, total mass and size are stored at the nodes of the subtrees. A representation of the domain decomposition algorithm is shown in figure 1.

Each of these subtrees must now be searched by each particle and the appropriate forces computed. This is done by first broadcasting all the tree data on one of the processors to all the other processors. The algorithm then proceeds by having all the particles on all the processors search this subtree which has just been broadcast to them. In other words, all particles are searching the same subtree at the same time. The algorithm is made complete by looping over all processors with a broadcast and search within the body of the loop. This algorithm is given in the following pseudocode.

DO i = 0, NPROCS - 1 *! NPROCS is the number of processors*

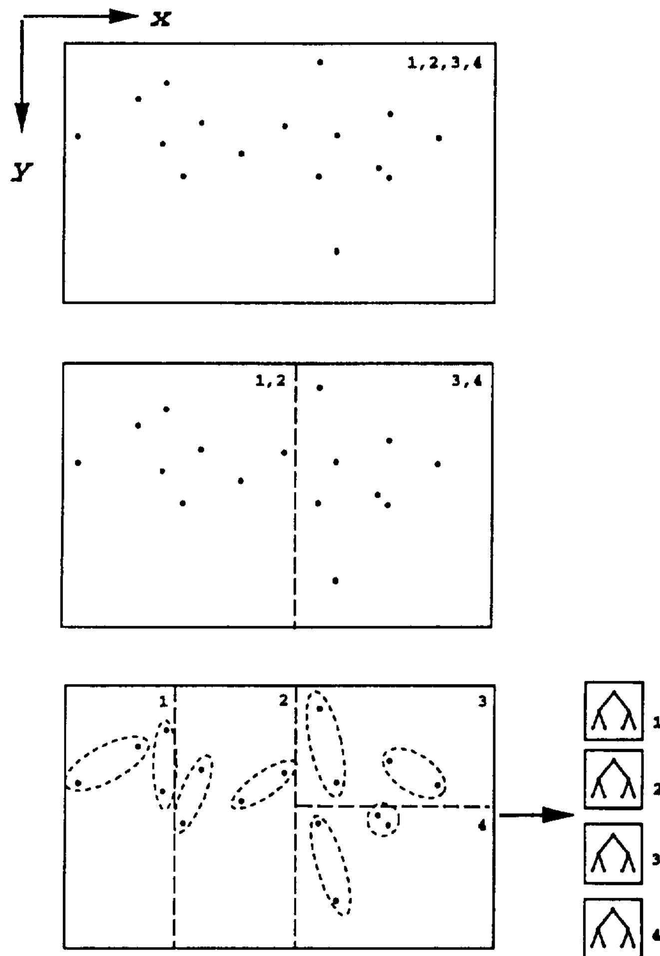


Figure 1: Schematic of the domain decomposition algorithm on a hypothetical four processor machine. The first panel shows the particles in a two dimensional space. The second panel shows the particles after one pass of the RBLD algorithm. The dashed line indicates the physical separtion which results by splitting the sorted list into two equal halves. The last panel shows the space after the final pass of the RBLD algorithm. The processor numbers associated with each domain are shown in the upper right corner of that domain. Note that after the final step the space was split along different dimensions. The dotted ovals indicate the particles which are grouped together in the first level of the each subtree and the small square boxes in the lower right corner of the figure show the balanced trees which are built on each processor (shown as binary trees for clarity).

```
    BROADCAST Tree Data from Processor 'i' to all Processors
    CALL SEARCH ! Particles on each processor search i'th tree

END DO
```

Now, since the particles have been sorted into processors to represent local regions of space and particles in different regions of space will have different, spatially dependent search path lengths through each individual subtree, the computational load of the algorithm as described above can become imbalanced for particle distributions which are irregular. We overcome this problem by first making copies of the particle data. We then apply the following algorithm on each processor in parallel to shuffle the copied particle data between processors:

```
ix = my_proc ! my_proc is the local processor number

DO j = 1,N/NPROCS ! N = number of particles

    if (ix.ne.j) then
      fetch particle j's data from processor ix
    end if

    ix = ix + 1
    if (ix.gt.NPROCS - 1) ix = 0

END DO
```

In this way each processor controls the tree searches for a set of particles which are distributed throughout the spatial domain of the simulation. Hence, the processors contain, on average, an equal amount of work since each searches the same subtree at the same time using a sampling of particles which represent the entire spatial domain of the calculation. The individual, on-processor tree searches were optimized for the Cray T3D by using an interaction list approach as described above and in [7]. Also, a routine written

in assembly language which computes the reciprocal square root was also utilized [14].

The scaling with processor number (N_p) and particle number N we would expect for this algorithm is then simply given by

$$t = K_1 \frac{N}{N_p} log(N) + K_2 N_p t_{broadcast}, \quad (3)$$

where $t_{broadcast}$ is the time to perform one broadcast and K_1 and K_2 are constants. If we further assume that $t_{broadcast}$ scales as $\frac{N}{N_p}$ (since the amount of data being broadcast per cycle of the loop scales as $\frac{N}{N_p}$). We arrive at the expected scaling of

$$t = K_1 \frac{N}{N_p} log(N) + K_2 N. \quad (4)$$

Hence, the time to solution of this algorithm scales linearly with processor number only if $K_1 \gg K_2$. This will only be true if the total time to do the broadcasts of the individual trees is small compared to the time to search that tree.

4 Performance

The above described algorithm was initially implemented using PVM [6] on a local experimental message passing architecture known as Beowulf [13]. The debugged code was then ported to the Cray Research T3D at the Jet Propulsion Laboratory. Where the message passing performance employing PVM was poor, the communication calls were replaced with the Cray shared memory calls (e.g. the broadcast of the subtrees). The code has been written entirely in Fortran and uses the default Cray 64 bit arithmetic.

To test the basic algorithm we have set up several test problems ranging from a highly uniform case to ones which have a high degree of clustering. To set up the uniform distribution the particle positions (x, y, and z coordinates) were simply chosen randomly from a uniform distribution between 0 and 1. To set up the clustered cases some number of clusters of particles, chosen by us, were randomly distributed in space. Each of these clusters was composed of an equal number of particles and the number density of particles as a function of radius within a cluster was chosen to vary as r^{-2}. A view of the resulting particle distribution for a case when the number of separate clusters was chosen to be 10 is shown in figure 2. Cases with

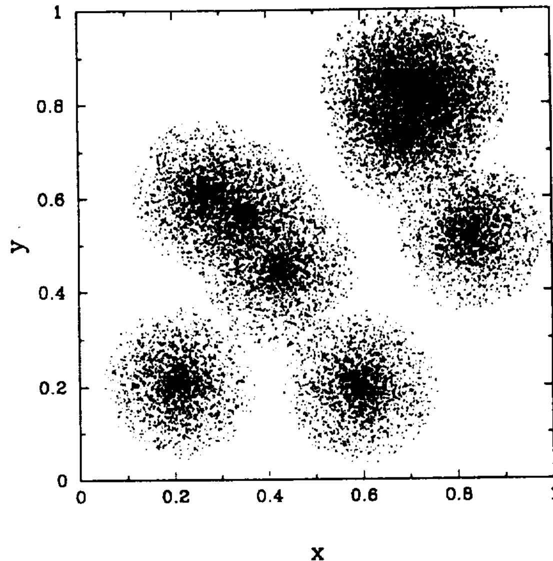


Figure 2: The particle distribution used for the tests of the tree searching algorithm. 32,000 particles are shown here and their positions have been projected onto the x, y plane. The density within each cluster varies with radius as r^{-2}.

1000 such clusters were also run. The performance results for the different particle distributions differed little and we report here only the performance measurements employing the 10 cluster case shown in the figure.

We first consider how the search algorithm described above scales with8q the number of processors. In figure 3 we show how the number of gravitational interactions computed per second scales with the number of processors. An interaction is defined to be the points during a tree search where an actual force computation is made on a searching particle. In the figure we show the results for 3 different problem sizes, 65,536 particles, 262,144 particles, and 1,048,576 particles. To obtain the number of interactions computed per second we took the total number of interactions computed and divided by the maximum of the times taken by each individual processor to perform their own tree searches. The tree search times include both the time to perform the on-processor tree searches plus the time to do the necessary broadcasts. From this figure we see that the scaling of the algorithm is better for larger

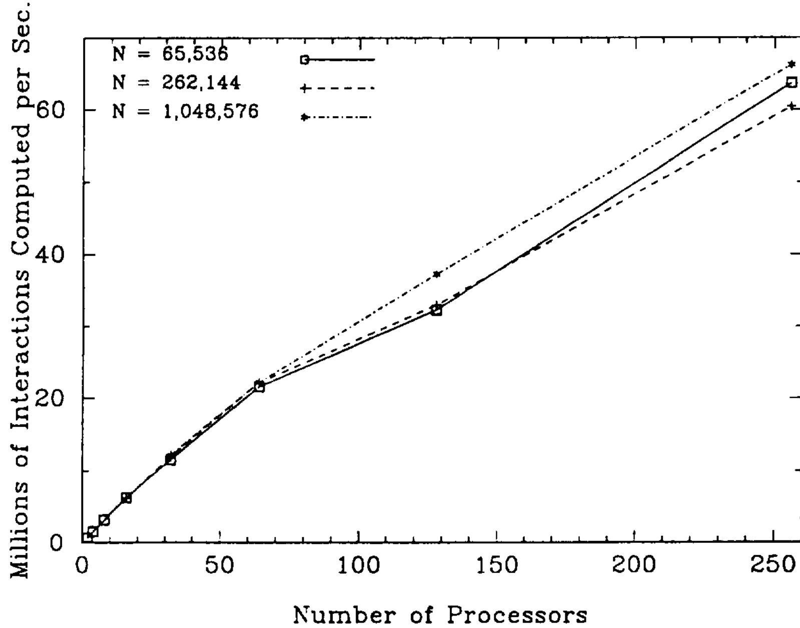


Figure 3: The number of interactions computed per second vs. the number of processors. The total number of interactions computed is a measure of the total amount of work that the algorithm performs independent of the number of processors and is divided by the total time to perform the tree search algorithm, including interprocessor communication, as described in the text. The tree search time for the entire algorithm is taken to be the maximum time among all the processors.

particle numbers. The times to execute using 256 processors which were measured for the 16,384 and 262,144 cases were 1.048 seconds and 5.54 seconds respectively. Therefore, for these cases the condition of $K_1 \gg K_2$ given above may not hold.

Speedups are plotted vs. the number of processors used in figure 4. For the 65,536 particle case speedups are computed relative the times measured using 2 processors. The speedups for the 262,144 particle cases and the the 256 processors cases are computed relative to the timings using 16 and 32 processors respectively. Here, we see that speedups are only near linear for processor numbers less than or equal to 64. Again, this indicates that the broadcasts of data becoming are probably a larger fraction of the cost for the cases with larger processor numbers.

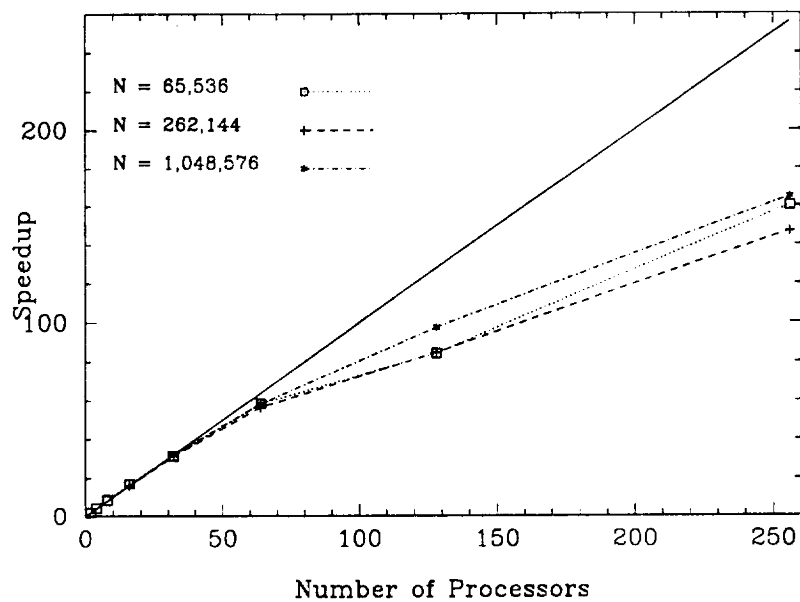


Figure 4: Speedup curves for the tree search algorithm described in the text. Ideal linear speedup is shown with the solid line. Results are shown for three different problem sizes as indicated. The curve for the 262,144 particle case was normalized relative to the result using 16 processors while the curve for the 1 million particle case is normalized relative to the result using 32 processors.

To estimate the useful floating point rate of the Cray T3D using the tree search algorithm described here we not that one gravitational interaction costs roughly 30 floating point operations. This gives floating point rates using 256 processors 1.9 Gflop/s, 1.8 Gflop/s, and 2. Gflop/s using 16,384, 262,144, and 1,048,576 particles respectively. Since the peak speed of a Cray T3D with 256 processors is near 15 Gflop/s, we clearly can do better. Improvements to the basic algorithm given above are discussed in the following sections.

5 Quadrupole Moments

Each node in the tree data structure need not be treated as a mass point as was done above. It has been shown that one can compute higher order moments of the mass distribution (i.e. quadrupole moments) and use them to increase the accuracy of the calculation. Further, the efficiency of the algorithm is improved since the addition of such high order terms does not affect the tree search and only adds additional floating point computations to each interaction list [7].

Assuming that the potential is softened as in equation 1 the traceless quadrupole moment tensor is not applicable and the potential at any point outside the sphere containing all the particles which belong to a tree node is approximated by,

$$\phi(\boldsymbol{r}) \simeq -G\left[\frac{M}{(r^2+\epsilon^2)^{1/2}} + \sum_{l=1}^{3}\sum_{k=1}^{3}\frac{1}{2}q_{k,l}\left(\frac{3x_k x_l}{(r^2+\epsilon^2)^{5/2}} - \frac{\delta_{kl}}{(r^2+\epsilon^2)^{3/2}}\right)\right], \quad (5)$$

where the observation point, $\boldsymbol{r}$, is measured with respect to the center of mass of the tree node (in which case the dipole moments are zero). The sums over k and l indicate sums over the 3 coordinate axes, M is the total mass of the node, δ_{kl} is the usual delta function ($\delta = 0$ if $k \neq l$, $\delta = 1$ if $k = l$), and $q_{k,l}$ are the quadrupole moments which are given by,

$$q_{k,l} = \sum_i m_i x_{i,k} x_{i,l}. \quad (6)$$

The sum is over all particles belonging to the tree node, m_i are the particle masses, and $x_{i,k}$ are their positions measured with respect to the center of mass of the tree node. The accelerations are then found from $\boldsymbol{a} = -\nabla\phi$.

During the tree build phase of the tree algorithm the quadrupole moments are computed for the mass distributions represented by each node in the tree

in much the same way as are the center of mass and the total mass. There are 6 distinct moments and they are represented by the sums over i in the above relation. The quadrupole moments at each level in the tree can be computed from those computed at a lower level (towards the leaves) rather than computing the sums over all particles by using the relation,

$$q_{kl}^{parent} = \sum_{children} q_{kl}^{child} + sum_{children} M^{child} x_k^{child} x_l^{child}, \tag{7}$$

where the sums are over the child nodes of a parent, M_{child} is the mass of the child node, and x^{child} are the centers of mass of the child nodes measured with respect to the center of mass of its parent node. Hence, the quadrupole moment of a parent node can be found from its children by first summing the quadrupole moments already computed for its child nodes and adding to this sum an additional set of terms which are the quadrupole moments computed from its child nodes treating them as if they were mass points.

Figure 5 shows the number of interactions computed per unit time as function of the number of processors for the case that the number particles was set to 265,144 and the value of θ was set to 1. In determining the total number of interactions it is assumed that each quadrupole interaction is equivalent to 6 monopole interactions. From this plot we see that the performance as measured by the number of interactions computed per unit time is indeed increased relative to the cases where quadrupole moments are not included. Again assuming 30 floating point operations per force interaction we arrive at a Mflop rating 5.4 Gflop/s for the case when 256 processors are used.

Speedup curves are shown in figure 6 for the same cases as those shown in figure 5. The speedups are somewhat better here than those shown in figure 4. This indicates that the additional floating point operations associated with the computation of the quadrupole interactions partially offsets the additional communications involved with broadcasting the additional quadrupole moment data.

Even though these performance levels are increased by the addition of quadrupole terms to the algorithm, we still need to consider overall running time of the algorithm as well as the accuracy reached within that time. For the case where 256 processors and 256,144 particles were used and quadrupole corrections were not included required 5.54 seconds to execute. The identical case which includes quadrupole corrections required 11.72 seconds.

As a measure of the accuracy of the algorithm we use the relative RMS

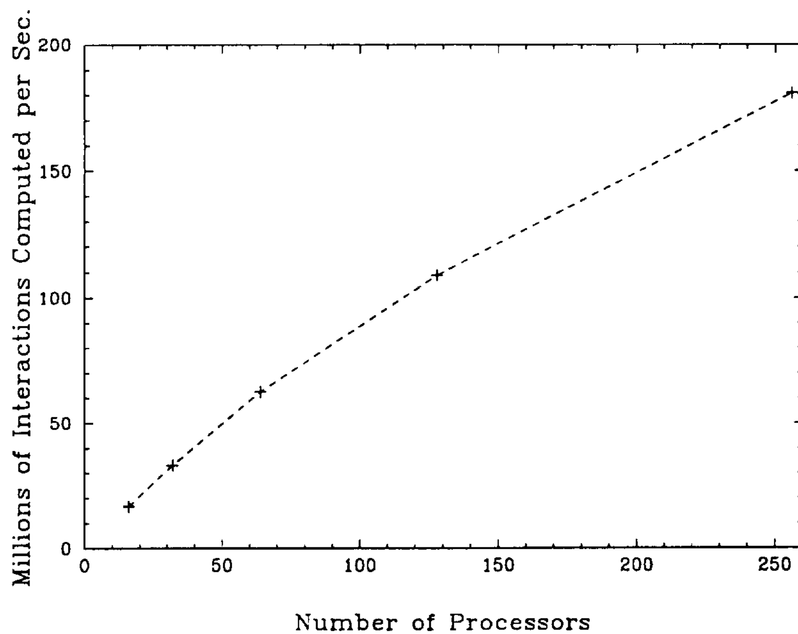


Figure 5: The number of interactions computed per second plotted vs. processor number when quadrupole corrections are included. The interactions are counted as described in the text. The number of particles used was 262,144 and θ was set to 1. The number of interactions per second using 256 processors corresponds to a performance of roughly 5.4 Gflop/s.

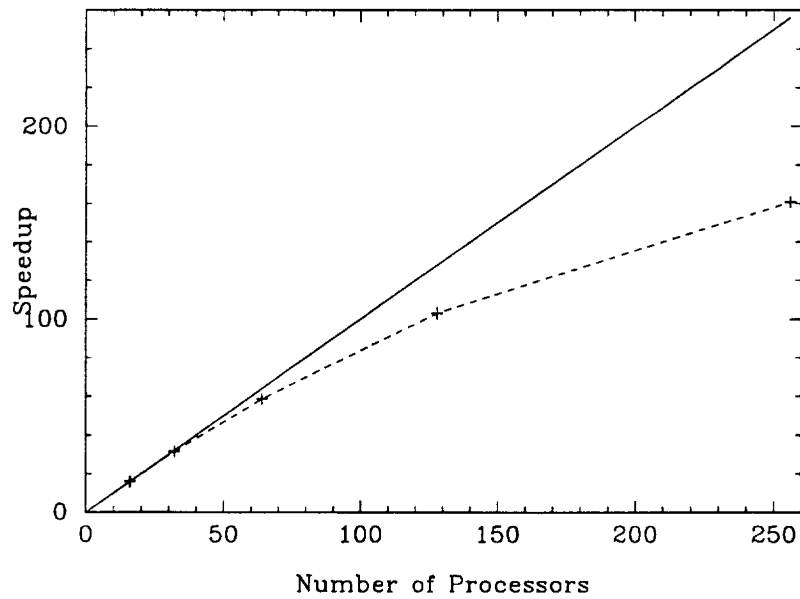


Figure 6: Speedup curve for cases when quadrupole corrections are included. The number of particles used was 262,144 and θ was set to 1. Results are normalized relative to the case when the number of processors is equal to 16.

error of the particle system. This is defined as,

$$RMS\ error = \sqrt{\sum_i \frac{|a^i_{exact} - a^i_{tree}|^2}{|a^i_{exact}|^2}}, \tag{8}$$

where a^i_{exact} represents the acceleration vector for particle i computed using the direct sum of all particle pair interactions and a^i_{tree} represents the acceleration vector for particle i computed using the tree. The two cases discussed above reached relative RMS errors of 5.99×10^{-2} for the case without quadrupole corrections and a relative RMS error of $4.98 \times 10-3$. Therefore, even though the case with quadrupole corrections requires roughly a factor of 2 more CPU time to execute, the accuracy achieved is at least a factor of 10 better.

6 Group Searching

Since particles which are spatially nearby one another will have search paths through the tree which are similar, arranging particles into spatial groupings which then search the tree would reduce the number of items which would need to traverse the tree data structure. This idea was first advocated by Barnes [3] and later applied to MIMD parallel machines by Dikiakos and Stadel [4] and by Olson [9] for SIMD machines.

The implementation of this idea used here exploits the sorted list of particles used to create the tree data structure. Here, all searching groups have equal numbers of particles and are created by dividing up the sorted particle list into sections of size n_{group}. For each group the center of mass is computed as well as a size of the group. As for the nodes in the tree we take the size of the group to be the maximum distance of the particles in that group as measured from the center of mass of that group.

Each of these groups then searches the tree in much the same way that individual particles searched the tree in the description given above. Here, however, the acceptance criterion must take into account the fact that the searching group of particles inhabits a finite region of space. For the implementation discussed here a node is accepted for the computation of a force if,

$$r > \frac{2S_{node}}{\theta} + S_{group}, \tag{9}$$

where r is the distance between the center of mass of the searching group and the center of mass of the tree node currently being visited, S_{node} is the

size of the tree node, S_{group} is the size of the searching group, and θ is a user supplied parameter which allows variable force accuracies. In the event that this condition is met, the data for the accepted node are placed in the interaction list. Once the tree search is complete for a searching group, forces are computed for all particles within that searching group by cycling through its interaction list.

The code has been written to allow the size (in particles) of the searching groups to be varied. Since we do not know what number of particles per searching group will result in the greatest efficiency, we attempt to arrive at its value empirically. Therefore, we plot in figure 7 the number of interactions computed per CPU time vs. the number of particles in each searching group. Here, results are shown for cases where the number of processors was 128, the number of particles was set to 262,144 and θ was set equal to 1. This plot shows that the peak in this curve occurs where the number of number of particles in a searching group equals 16. This peak corresponds to a performance number of 4.1 Gflop/s.

The results shown in figure 7 suggest that the optimal value for the number of particles per searching is 16 (although it is only marginally better than some of the other cases shown using this measure of performance). Still, we must consider whether the running time of the code relative to the force accuracy achieved is improved as well. Therefore, we plot in figure 8 the running time of the tree search and force computation vs. the logarithm of the RMS error. The lines in the plot are for values of the number of particles per searching group of 1, 4, 16 and 64. To obtain different values of the accuracy θ was varied between 1. and .4. Again, 128 processors were used in each case. This plot shows that the 16 and 64 particles per searching group cases give the best results and that they are virtually identical and are only marginally better than the case where the number of particles per searching group equals 4.

7 Loop Blocking

Due to the small size of the on-processor cache of the T3D, we may expect different performance levels if do loops are written so that they can be blocked. To do this for the code considered here, we vary the size of the interaction list by never allowing its size to get larger than a pre-chosen value. When this number is exceeded the tree search is temporarily stopped and the interaction list is flushed. Several cases were run which included quadrupole

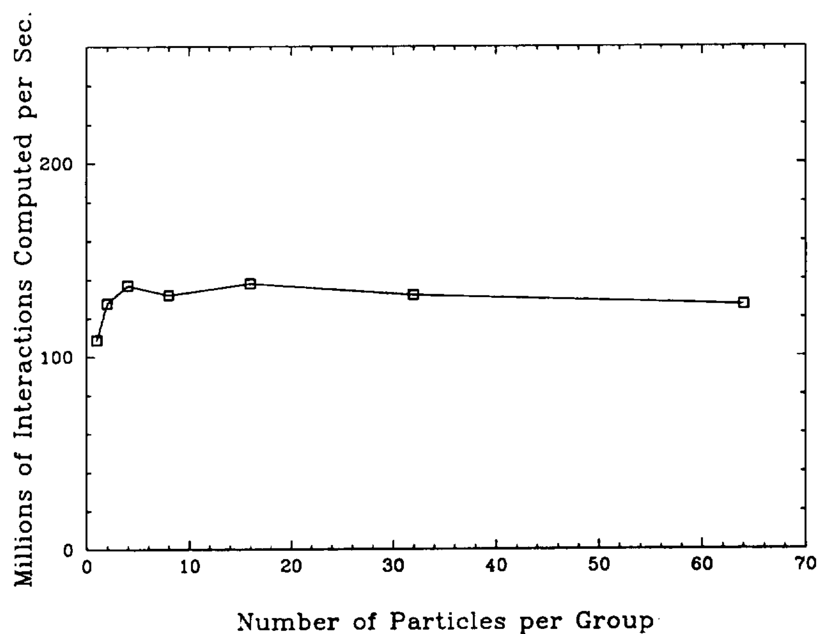


Figure 7: Number of Interactions computed per second plotted vs. the number of particles per searching group. The number of particles used was 262,144 and θ was set to 1. The number of processors was held fixed for each case at 128.

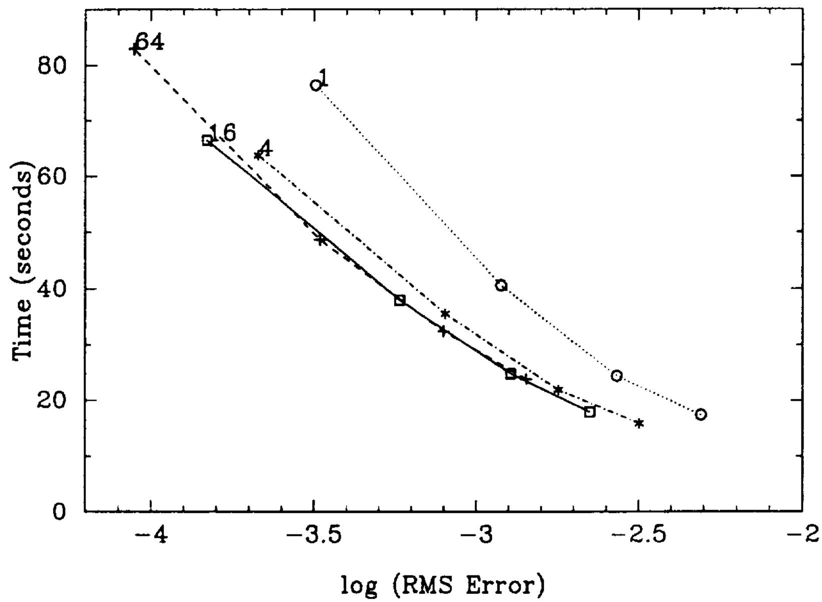


Figure 8: Running times of the tree search and force computation algorithm plotted vs. the logarithm of the relative RMS error. The number of particles used was 262,144 and the number of processors used 128. The different curves indicate results for different values of the number of particles per searching group as indicated by the number associated with each curve.

corrections and the group searching algorithm described above. The value of θ was set to 1. It was found empirically that the best results are achieved when the maximum length of the interaction list is ~ 25. For this value the code computed 167 million interactions per second (5.0 Gflop/s) using 128 processors. This is a modest improvement of a factor of 1.2.

8 Scaling

As we commmented above, we do not expect the scaling of the algorithm to be linear for indefinitely large number of processors. This seems to be born out be the results presented above. In other words, the time spent in the individual searches of the trees is reduced as the processor number is increased, but the time to do the broadcast in the algorithm remains roughly constant. We find that timings of the broadcast overhead bear out this conjecture. Cases identical to those shown in figure 6 were run and the overhead associated with the broadcast was timed separately. Cases using from 16 up to 256 processors were used. The broadcast time varied little from case to case and ranged from values of 2.35 seconds using 16 processors to 2.77 seconds using 256 processors. Figure 9 shows the scaling curve which results by subtracting off the broadcast times from the total running times of the tree search as compared to the scaling of the total time to execute the tree search algorithm. This plot indicates that while the broadcast is a major reason for the lack of linear scaling, other factors prevent further improvement. The major reason for this is probably algorithmic. Since each tree on each processor becomes smaller as the number of processors is increased, each searching item will in effect search deeper in the overall tree structure than it otherwise would need to as compared to the equivalent problem run on a single processor.

We further note that one way to get better scaling of the overall tree search algorithm is to give more work to each on processor tree search. This can be accomplished by setting θ to a smaller value. Figure 10 shows scaling curves for cases where θ was set to values of 1 (long dashed curve) and .6 (short dashed curve). The performance at 256 processors and $\theta = .6$ corresponds to a real performance level of 9.6 Gflop/s.

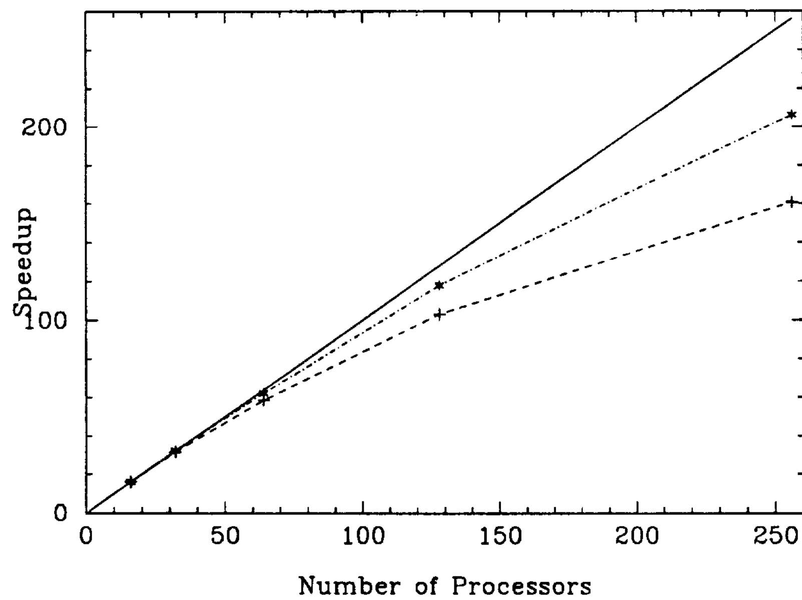


Figure 9: Scaling curves with and without broadcast times included. The long dashed curve is the same as in figure 7 while the short dashed curve does not include the broadcast time.

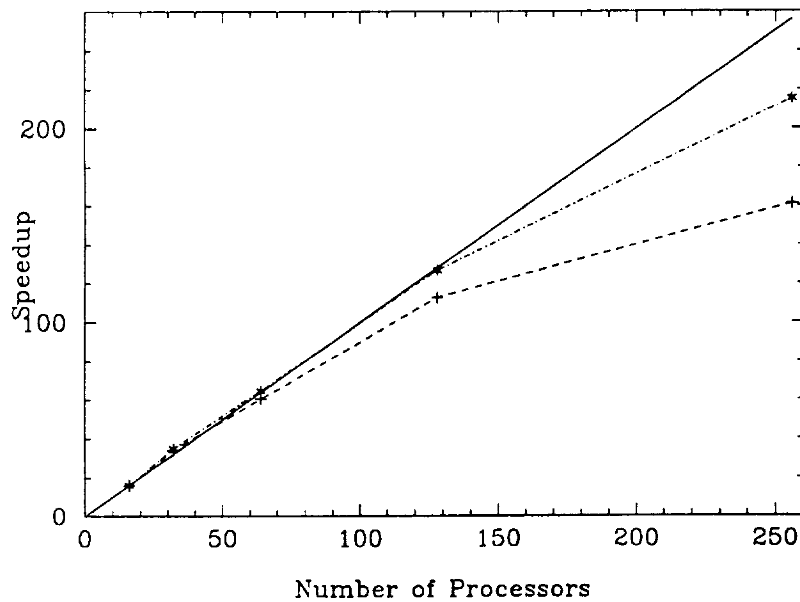


Figure 10: Scaling curves using different accuracies. The long dashed curve shows results using $\theta = 1$ and the short dashed curve shows results using $\theta = .6$. 262,144 particles were used, quadrupole corrections were included and the group searching algorithm with 16 particles per searching group was used. The curves are normalized to results obtained at 16 processors.

9 Conclusion

We have described an extremely simple algorithm for implementing a gravitational N-body tree code on a coarse-grained parallel computer architecture. Performance measurements of the algorithm have also been taken using the Cray Research T3D parallel computer located at the Jet Propulsion Laboratory. The algorithm relies heavily on interprocessor broadcasts of data, and scaling with the number of processors is shown to be affected by this. Still, overall real performance of the code is good and roughly 9.5 Gflop/s are achieved using 256 processors of the Cray T3D.

This research was supported, in part, by NASA grant NAG5-2652 to George Mason University and also NASA's High Performance Computing and Communications Initiative.

References

1. Appel, A. W. An Efficient Program for Many-Body Simulation. *SIAM J. Sci. Stat. Comput.* **6**, (1985), 85-103.
2. Barnes, J. E., and Hut, P. A hierarchical O(Nlog(N)) force-calculation algorithm. *Nature* **324**, (1986), 446-449.
3. Barnes, J. E. A Modified Tree Code: Don't Laugh, it Runs *J. Comput. Phys.* **87** (1990), 161-170.
4. Dikaiakos, M. D. and J. S. A Performance Study of Cosmological Simulations on Message-Passing and Shared-Memory Multiprocessors' (1995), submitted to the International Conference on Supercomputing '96, (http://www-hpcc.astro.washington.edu/papers/marios/perform/perform.html)
5. Dorband, J. E. Sort computation. *Proc. Frontiers of Massively Parallel Computation.* IEEE Computer Society, Washington D. C., 1988, 137-141.
6. Geist, A., Beguelin, A., Dongarra, J., Weicheng, J., Manchek, R., and Sunderam, V. *Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, 1994.
7. Hernquist, L. Vectorization of tree traversals. *J. Comput. Phys.* **87**, (1990), 137-147.
8. Jernigan, J. G. and Porter, D. H. A tree code with logarithmic reduction of force terms, hierarchical regularization of all variables, and explicit accuracy controls. *Astrophys. J. Suppl.* **71**, (1989), 871-893.
9. Olson, K. M. Efficient Tree Code on SIMD Computer Architectures. submitted.
10. Olson, K. M. and Dorband, J. E. An Implementation of a Tree Code on a SIMD Parallel Computer. *Astrophys. J. Suppl.* **94** (1994), 117-125.
11. Salmon, J. K. Parallel hierarchical N-body methods. Ph.D. thesis, California Institute of Technology, 1991.
12. Salmon, J. K. and Warren, M. S. Skeletons from the treecode closest. *J. Comput. Phys.* **111**, (1994), 136-155.
13. Sterling, T., Becker, D. J., Savarese. D, Dorband, J. E., Ranawake, U. A. and Packer, C. V. BEOWULF: a parallel workstation for scientific computation. *Proc. of the International Conference on Parallel Processing.* submitted, 1995.

14. Ranawake, U. A. V_SQRT: A Library Routine for Computing Square Roots of Double-Precision Vectors on Cray MPP
(software available at: http://sdcd.gsfc.nasa.gov/ESS/exchange/contrib/udaya/v_sqrt.html)

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
May 1996

3. REPORT TYPE AND DATES COVERED
Contractor Report

4. TITLE AND SUBTITLE
An N-body Tree Algorithm for the Cray T3D

5. FUNDING NUMBERS
G-NAG5-2652
Code 934

6. AUTHOR(S)
Kevin M. Olson and Charles V. Packer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS (ES)
Goddard Space Flight Center
Greenbelt, Maryland 20771

8. PEFORMING ORGANIZATION REPORT NUMBER
96B00085

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS (ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
NASA CR-199882

11. SUPPLEMENTARY NOTES
Olson: Institute for Computational Science and Informatics, George Mason University, 4400 University Drive, Fairfax, Virginia; Packer: Hughes STX Corporation,4400 Forbes Blvd., Lanham, Maryland.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 90

Availability: NASA CASI (301) 621-0390.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

We describe in this paper an algorithm for solving the gravitational N-body problem using tree data structures on the Cray T3D parallel supercomputer. This implementation is an adaptation of previous work where this problem was solved using an SIMD, fine-grained parallel computer. We show here that this approach lends itself, with small modifications, to more coarse-grained parallelism as well. We also show that the performance of the algorithm on the Cray T3D parallel architecture scales adequately with the number of processors (up to 256). Specific levels to be reached using the Cray T3D parallel architecture. A peak performance level of 9.6 Gflop/s is reached on 256 processors for the time critical gravity computation.

14. SUBJECT TERMS
Computational Techniques, Particle Methods, Gravitation, Parallelization

15. NUMBER OF PAGES
36

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

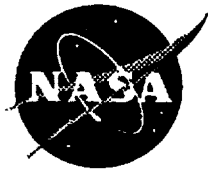
Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39.18

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

Official Business
Penalty for Private Use, $300

SPECIAL FOURTH-CLASS RATE
POSTAGE & FEES PAID
NASA
PERMIT No. G27



POSTMASTER: If Undeliverable (Section 158, Postal Manual) Do Not Return