A PARALLEL ADAPTIVE FAST MULTIPOLE ALGORITHM FOR N-BODY PROBLEMS*

Sanjeev Krishnan and Laxmikant V. Kalé
Department of Computer Science
University of Illinois
Urbana, IL 61801.
E-mail: {sanjeev,kale}@cs.uiuc.edu

Abstract

We describe the design and implementation of a parallel adaptive fast multipole algorithm (AFMA) for N-body problems. Our AFMA algorithm can organize particles in cells of arbitrary shape. This simplifies its parallelization, so that good locality and load balance are both easily achieved. We describe a tighter well-separatedness criterion, and improved techniques for constructing the AFMA tree. We describe how to avoid redundant computation of pair-wise interactions while maintaining load balance, using a fast edge-partitioning algorithm. The AFMA algorithm is designed in an object oriented, message-driven manner, allowing latency tolerance by overlapping computation and communication easily. It also incorporates several optimizations for message prioritization and communication reduction. Preliminary performance results of our implementation using the Charm++ parallel programming system are presented.

1 Introduction

The N-body problem is an important core problem arising in several simulation applications in astrophysics, molecular dynamics and fluid dynamics. The computation in the N-body problem consists of calculating interactions between all pairs of bodies (particles) in the system. The order of complexity for a simple formulation of this problem would thus be $O(N^2)$, for a system consisting of N particles. There exist many good algorithms for solving the N-body problem, of which the Fast Multipole Algorithm due to Greengard and Rokhlin [?] has $O(N)$ time complexity for N particles, for any desired bound on error. Other algorithms include the $O(NlogN)$ Barnes-Hut method, particle-in-cell methods, and the Distance Class methods. Although the problem sizes at which the Fast Multipole method would outperform the Barnes-Hut method is not clear, there has recently been a lot of interest in implementing both algorithms on parallel machines.

In the FMA, the effect due to a *well-separated* (sufficiently far away) group of particles is approximated by a *multipole expansion*, which is a refined formalization of the center-of-mass, leading to provable error bounds. Interactions are computed between particles and groups of particles, as well as between different groups of particles. In order to partition the particle set into groups, the FMA recursively divides the computational space into cubical "cells", which are ordered hierarchically in a tree. For the three dimensional problem an octtree is generated. In the *non-adaptive FMA*, a uniform grid is imposed on the computational space, resulting in a complete tree whose leaves all have the same depth. However, this is unsuitable for non-uniform particle distributions. Hence the *adaptive FMA* [?] divides cells until the number of particles in a leaf cell is less than some specified *grain-size*, leading to an irregular tree.

The adaptive FMA (AFMA) is difficult to efficiently program on parallel computers, especially for private memory ones. This is because of the difficulty of achieving good load balance and locality simultaneously, distributing shared data structures such as the tree among processors, overcoming latencies arising due to the inherent irregularity and unpredictability of the computation, and reducing communication volume. There are parallel implementations of the non-adaptive FMA on shared and distributed memory computers, including the work by Board and others [?, ?], and of the adaptive FMA on shared memory computers [?]. Our work is one of the first parallel implementations of the AFMA on message passing computers.

2 Modifications to the original AFMA

In the original AFMA [?], tree construction proceeds by recursively subdividing the computational space (which is initially cubical at the outermost level) into smaller cubical boxes of the same size (dimensions) : 4 boxes for the two-dimensional case and 8 in three dimensions. All boxes at the same level in the tree are the same size. For adaptiveness, the decision to subdivide a box depends on the number of particles it contains, so that the resulting tree is irregular.

The primary difficulty the original AFMA poses, especially for parallel implementations, is that the *partitioning of space is dictated by the AFMA algorithm*, and cannot be determined by either the parallel partitioning strategy or by the application using the AFMA code. While there is a

*This research was supported in part by the National Science Foundation grants CCR-90-07195 and CCR-91-06608.

lot of research into good libraries for parallel partitioning techniques and heuristics (which achieve good load balance and minimal communication), none of these techniques can be used because of the cubical subdivision of space the original AFMA requires. Instead new constrained techniques have to be developed, such as the costzones scheme [?]. These special techniques can give good load balance, but degrade locality since they cannot ensure that the regions mapped to processors are convex in shape.

Again, the application itself often has its own requirements for partitioning : e.g. in a molecular dynamics application NAMD [?] used in a Grand Challenge applications group at the University of Illinois, the partitioning of atoms into cells is determined by cutoff distances for other force calculations. In order to use the AFMA for long range Coulomb force calculations, the atoms have to be redistributed before AFMA and brought back to their original cells after the AFMA step, incurring significant overhead both in terms of performance and programmability.

To overcome these problems, we have developed a modified version of the FMA which does not require subdivision into cubical boxes (note that cubical subdivision is not theoretically required). In the modified algorithm, the computational space can be divided into *regions of any shape*, as determined by the partitioning algorithm or the application itself. In our implementation we have used orthogonal bisection for partitioning[1], in which each processor gets a contiguous, convex (rectangular) region of space, which ensures good load balance and low inter-processor communication.

In the original AFMA, two boxes of the same size are said to be well-separated if they are separated by a distance greater than their width. Implementations in three dimensions [?, ?] require the distance between well-separated boxes to be twice their width. Well-separation is required in order to maintain error bounds for the translation operators for multipole expansions. To handle boxes of arbitrary shapes and sizes, we now define a slightly different, more general well-separatedness criterion using Greengard's original theorem [?]. C_1 and C_2 are two cells, having r_1 and r_2 as their radii (distance from center to farthest particle), and z_1 and z_2 as their centers. We say that C_1 is *partially well separated* from C_2 if $|z_1 - z_2| > cr_1 + r_2$, where c is a constant greater than 1. This corresponds to a distance of cr_1 from the center of C_1 to the nearest particle in C_2. Similarly, C_2 is partially well separated from C_1 if $|z_1 - z_2| > cr_2 + r_1$. We say that C_1 and C_2 are well-separated if they are both partially well-separated from each other, i.e. $|z_1 - z_2| > max\{cr_1 + r_2, cr_2 + r_1\}$.

Using the above definition of well-separatedness, the four types of interaction lists for C_1 in the AFMA [?] can be defined as :
- V list : A cell C_2 is in the V list of C_1 if C_1 and C_2 are well-separated. Then C_2's multipole expansion can be converted to a *local expansion* at C_1's center.
- W list (leaf cells only) : if C_1 and C_2 are not well-separated, but C_2 is partially well-separated from C_1, and C_1 is a leaf cell, then C_2's multipole expansion needs to be evaluated directly at C_1's particles.
- X list : if C_1 and C_2 are not well-separated, but C_1 is partially well-separated from C_2, and C_2 is a leaf cell, then individual particle fields from C_2 need to be converted to a local expansion at C_1's center.
- U list : (leaf cells only) if none of the above three conditions apply between C_1 and C_2, then interactions have to be computed directly, between each pair of their particles.

In the original non-adaptive FMA, the well-separation criterion between spherical regions is applied to the cubical boxes case by requiring two intervening boxes (for 3-D) between the boxes being compared [?]. This is clearly an overkill, since for example, corner boxes in a cubical region are more distant from the center than other boxes. Intuitively, the volume enclosed by a cube is greater than the the volume enclosed by a sphere inscribed in the cube. By comparing boxes for well-separation using a tighter spherical criterion, there are fewer of the expensive pair-wise interactions. E.g in the three dimensional case, instead of computing 125 interactions with neighboring cells at the leaf level with the cubical criterion, there are only 93 interactions with our tighter criterion.

We have also developed an optimization for formation of interaction lists for cells. In the original FMA, the interaction list for a cell C consists of those cells *at the same level in the tree* which are well separated from C, but whose parents are not well separated from C's parent. Thus comparisons for well-separatedness are always between cells of the same size and hence at the same level of the tree. Instead, if we use our more general well-separatedness criterion, we can compare cells at different levels in the tree. E.g. C and D are two non-leaf cells, with 8 children each. If they are not well separated, there will be totally 64 interactions (for each pair of children). However, if C is well separated from 6 of D's children, then those 6 interactions can be computed directly, and the remaining pair-wise, resulting in only $6 + 8 * 2 = 22$ interactions. Thus we can further reduce the number of interactions that need to be computed.

3 Partitioning and tree construction

We first describe a fast parallel partitioning algorithm. We use orthogonal recursive bisection (ORB), which is a well known technique which recursively partitions the computational space by planes parallel to the coordinate axes. The load measure for partitioning is obtained from the previous AFMA iteration by a scheme similar to [?]. The partitions resulting from ORB are rectangular, resulting in less communication with neighboring partitions. Also, since partitions are not restricted to be cubical, much better load balance can be achieved. If there are P processors, the number of parallel bisection operations needed is usually $P - 1$. However, all these bisections need not be serialized. In fact, the critical path involves only $log_2 P$ bisections, corresponding to the height of the binary tree formed by hierarchically ordering the partitions. Hence we

[1] We emphasize, however, that in our modified AFMA, the partitioning strategy used can be quite general.

overlap the parallel bisection operations to significantly reduce idle times of all processors. This is done with little programming effort because of the message-driven model of our design (see section 5).

At the end of partitioning, processors exchange particles so that each processor has its own set of particles. A copy of the top $log_2 P$ levels of the AFMA tree is also made on all processors, so that all processors have the root cell of all other processors' local trees. Now each processor proceeds to construct its own local part of the tree by recursively dividing the space assigned to it by ORB. The division continues till the load on each leaf cell is less than some prespecified grainsize. After tree construction is completed, interaction lists for the shared (top $log_2 P$ levels) and local trees are computed.

After each processor builds its local tree, it needs to get parts of other processors' trees in order to compute the remote members of the interaction lists required for the AFMA. The purpose of this step is similar to the Locally Essential Tree (LET) construction step in [?], however, we avoid the overheads associated with their implementation by two optimizations. First, we assemble only the structure of the LET and the coordinates for each cell; the particles and multipoles for each cell are not transferred because they are not required for interaction list formation. Second, instead of fine-grained receiver initiated communication for expanding the tree one level at a time [?], each processor sends to its neighbors (the processors it needs to interact with, as determined by the list-formation algorithm) the *exact* part of its own local tree that they will require, resulting in just one message per neighbor. The key insight that allows us to do this for the AFMA is the following : if a local cell C is partially well separated from the root cell for processor P, then from the definition, C must appear in the V or W lists of all cells in P, and hence C's children do not need to be sent to P. Thus we identify all cells that need to be sent to neighboring processors using just the shared top levels of the AFMA tree.

Once each processor has received parts of the LET from other processors (no exchange of the LET is needed with well-separated processors), and attached them to its copy of the shared top levels of the tree, it has the entire LET, and can start computing members of the four lists for each of its cells. The list forming algorithm consists of a recursive depth first traversal of the tree starting at the root cell of the tree. Each cell has a list of "candidate cells" which is built while traversing its parents. For internal cells in the tree, the algorithm tries to classify candidate cells into either the V or X lists. Those cells that cannot be classified are added to the candidate-lists of all child cells, so that they can be classified at a lower (finer) level in the tree. Finally, at the leaf cells, all remaining candidate cells are added to the U or W lists. Details of this algorithm are not described here due to space limitations.

4 Balancing pair-wise interactions

Pair-wise (U list) interactions take up a major portion of the time for computing interactions between cells. To eliminate redundant pair-wise interactions, Newton's third law is usually used. For a pair of cells A and B, particle data is sent from A to B, where the forces are computed and sent back to A. Although this insight for removing redundant computation is well known, in the parallel context we are faced with the problem of deciding, for every U list computation, which processor it should be computed on. If U list computations are not assigned to processors carefully, it is possible that some processors will get overloaded.

We can arrive at a heuristic solution to this load balancing problem by formulating it as an *edge-partitioning problem* on a graph. The vertices of the graph are processors. An edge is drawn between a pair of processors P and Q if there is a U list interaction between a cell on P and a cell on Q. The *weight* of this edge is the total computation cost of all cross-processor U list interactions between cells on P and cells on Q. The problem now reduces to that of partitioning the load of each edge among the processors it connects such that all processors have approximately equal loads after all edges have been assigned. Although this edge-partitioning problem can be solved optimally, we require a fast, easily parallelizable algorithm.

We have developed a heuristic algorithm by making use of *spatial information* which is already present in the AFMA tree. This spatial information is in the form of the repeated bisections performed while constructing the shared levels of the tree. The main idea in our edge-partitioning algorithm is hence to recursively divide the edge loads between sibling cells (which represent adjacent partitions) in the hierarchical tree. The heuristic for dividing cross-partition edges between two partitions takes into account local as well as global criteria. The global requirement is that the loads of the two partitions should be balanced. The local requirement is that the edge should not be assigned to a processor which is already heavily loaded. We have obtained excellent results by using the local criterion when processor loads are close to the average load (as would be the case towards the end of this algorithm), and using the global criterion otherwise. Each processor recursively bisects *only the subtrees it belongs to.* If E is the total number of edges and V is the number of processors, then in the best case each processor has to examine $E + E/2 + E/4 + ...$ edges, which means that the algorithm has a best case complexity of $O(E)$ and a worst case complexity of $O(E * logV)$[2]. In practice the running time is close to the best case because the initial ORB partitioning of particles ensures that the number of edges across partitions is balanced to some extent. Together with the ORB partitioning, this edge-partitioning heuristic results in good load balance, as well as good locality.

5 Overlapping communication latency

We have implemented our algorithm in the Charm++ parallel programming language [?, ?], which is an extension of C++. Charm++ has a *message driven model*, in which functions inside objects are invoked in response to

[2] Optimal algorithms for this have a complexity of $O(V * E)$ or more.

the receipt of messages; there are no explicit blocking "receive" calls. Cells in the AFMA tree are modeled as objects having functions for computing the different interactions. Thus the design consists of concurrently executing objects which interact by asynchronous function invocations.

As discussed in section 3, the partitioning by ORB can be optimized by performing bisections concurrently. This allows idle times during one bisection to be overlapped with computation from the other. Again, during the construction of the Locally Essential Tree, it is not necessary for a processor to wait for the parts of the tree from other processors ; it can continue with the construction of its own local tree. When a message containing tree-data from another processor arrives, the code for processing it is scheduled and that part of the tree is attached to the local tree.

Each of the four types of interactions (corresponding to the four lists) involve highly parallel and irregular interactions between cells in the AFMA tree. For each of the interactions, all processors iterate over their cells, computing local interactions and sending/receiving messages for remote ones. Interactions with remote processors are computed completely asynchronously: when a message containing remote data arrives, the object to which it is directed is automatically scheduled by the Charm runtime system, and the proper function for computing the interaction is invoked. Thus no time is wasted in waiting for remote data, and an almost ideal overlap of communication and computation is achieved.

Although there is much parallelism within each stage of the AFMA, executing them sequentially (as has been done in almost all previous implementations) can lead to serious imbalances and processor idling. This is because it is difficult to balance the load in each stage by itself. In [?] an attempt has been made to explicitly overlap two stages (in the context of the Barnes-Hut method) using a "non-synchronizing" global communication protocol. However, this requires some programming effort, and is difficult to achieve when there are many stages with complex dependences.

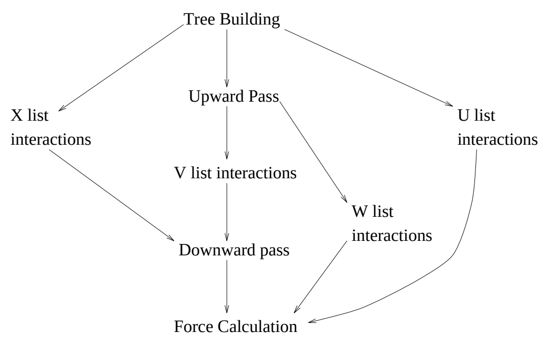

Figure 1: Dependences between stages of the AFMA.

Figure 1 shows the dependences across the various stages of the AFMA. It can be seen that there is a lot of scope for overlapping the idle times in one stage with computation from other stages. This is achieved naturally, and with *no extra programming effort*, because of the asynchronous, message driven nature of our design. Each processor just starts off all the interactions, and then dependences between stages are enforced *at the granularity of the cell objects*, thus avoiding any need for global synchronization between stages. This overlap across stages enables us to prevent processors from idling when there is work in other stages left.

6 Priorities and other optimizations

From Figure 1, it is clear that the dependence between stages is complex, and needs to be exploited carefully. There is a significant critical path consisting of tree construction, the upward pass in the AFMA tree, the V-list interactions, the downward pass in the tree, and finally the force evaluations. On the other hand, U-list interactions can proceed independently for all cells, and depend on only the tree construction to complete. Hence they can be thought of as "background computations" running at lower priority, which are used to fill up idle times during the other stages. We use the *message prioritization* feature of the Charm run-time system [?] to give the highest priority to messages traversing up and down the tree, while the V-, X-, W- and U-list interactions are given successively lower priorities. This enables the scheduler on each processor to always schedule computations on the critical path ahead of the others.

The communication patterns in the parallel AFMA are unpredictable and irregular. This is especially a problem for distributed memory computers which have high message latencies. Thus it is not possible to get good performance with fine-grained, receiver initiated communication, as is possible for shared memory machines [?]. We have extensively used a sender-initiated *advance-send* protocol to reduce communication overhead in our implementation. In [?, ?] a form of advance-send is used in the context of the Barnes-Hut method by sending particle data to remote nodes instead of requesting for their particles. For the AFMA, the basic problem to be solved before using advance-send is for each processor to determine which other processors will be consumers of its particle and multipole data. The key insight is to find, for each interaction list, the *dual interaction list.* From the definition of the four lists, we find that the U list is its own dual, the V list is its own dual, and the W and X lists are duals of each other. Thus each cell simply has to advance-send its particles to all cells in its U list, its multipole expansions to all V and X list cells, and a converted local expansion to all W list cells. In the AFMA, sending multipole expansions instead of particles has the advantage that a single multipole expansion is usually much less voluminous than the particle data itself, for cells containing tens or hundreds of particles. As described earlier, when a message carrying this data arrives at a processor, the Charm run-time automatically schedules the correct interactions in the appropriate cell objects. Thus advance-send is achieved with almost no programming effort.

To increase the granularity of communication, it is necessary to combine or aggregate messages going to the same destination processor. All-to-all personalized communication is also required during the partitioning stage. We have implemented library classes for these operations, which are used by simply inheriting a message class from them.

7 Preliminary performance results

We have implemented our parallel AFMA algorithm using the Charm++ portable object-oriented parallel programming system. Table 1 presents preliminary results on the TMC CM-5 and Intel Paragon for our 3-D AFMA implementation. The timings are for a nonuniform distribution[3] of 20,000 particles, for one time-step. The number of multipole expansion terms is 8, corresponding to the high-accuracy simulations in the work by Board [?]. The results do not include the parallel tree formation step. It was observed that several computation steps can be carried out before a tree formation is required; also a new tree-partitioning every time-step is not required for the correctness of the algorithm, as long as particles are moved to their new cells after a computation step. A detailed analysis of performance is not presented here due to lack of space.

Processors	16	32	64	128
Paragon	138.3	72.8	37.8	19.7
CM-5	315.4	147.1	74.9	42.2

Table 1: Time (in seconds) to simulate one time-step for 20,000 particles on the Intel Paragon and TMC CM-5.

8 Summary

In summary, the contributions of this work are :

- Our work is one of the first parallel implementations of the adaptive fast multipole algorithm on distributed memory computers.
- The algorithm is based on a modified version of the Greengard-Rokhlin AFMA; it allows particles to be organized in cells of arbitrary shape, instead of restricting them to be cubical. This simplifies parallel partitioning, so that good locality and load balance are both easily achieved.
- We have developed a tighter well-separatedness criterion which reduces the number of expensive pair-wise interactions, and an improved technique for constructing the locally essential tree.
- We have designed a fast edge-partitioning heuristic to solve the load-balancing problem caused when redundant pair-wise interactions are eliminated.
- Our implementation automatically overlaps computation and communication within and across stages of the algorithm.
- Our implementation is optimized using message prioritization, advance-sends, and other communication optimizations.

Acknowledgements

We would like to thank John Board at Duke University for the sequential FMA code on which our implementation is based.

[3] Each coordinate value in this distribution was generated by doing a bitwise AND of two random integers, and then normalizing it within the computational box. Thus most particles are concentrated at one of the corners of the box.